SMART STORAGE FOR SMART MOBILE DEVICES

A Dissertation
Presented to
The Academic Faculty

By

Ashish Bijlani

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of School of Computer Science

December 2020

SMART STORAGE FOR SMART MOBILE DEVICES

Approved by:

Professor Umakishore Ramachandran, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Mostafa Ammar
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Professor Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Professor Raghupathy Sivakumar
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: December 4, 2020

To my wife and best friend,

my soon-to-be-born son,

my parents,

and my sister,

for their love and support.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Professor Umakishore Ramachandran for his unwavering support, guidance, encouragement, and mentorship in my academic and entrepreneurial endeavors.

I would like to thank the committee members, Professor Mostafa Ammar, Professor Ada Gavrilovska, Professor Vivek Sarkar, Professor Raghupathy Sivakumar for taking their precious time to serve on my thesis committee, and providing their insightful feedback that significantly improved this thesis.

I would like to express my appreciation and gratitude to Professor Roy Campbell and Professor Taesoo Kim for providing me an opportunity to work with them and mentoring me as I was learning to carry out academic research.

I am forever grateful to late Professor Karsten Schwan and Professor Ada Gavrilovska for providing me my first academic research opportunity, and supporting me throughout my Masters. The experience instilled a desire to pursue doctorate.

I am thankful to my amazingly talented colleagues and collaborators, Ruian Duan, Yang Ji, Sangho Lee, Simon Pak, Chengyu Song, Hong Hu, Sanidhya Kashyap, Mohan Kumar, Steffan Maas, Adam Hall, Harshit Gupta, Enrique Saurez, Zhuangdi (Andy) Xu, Read Sprabery, Sayed Hadi Hashemi, Shayan Saeed, Mohammad Ahmad, Imani Palmar, Faraz Faghri, Faria Kalim, and Shadi Noghabi.

I would also like to extend my gratitude to my mentors Jeff Garbers and Melissa Heffner from Venture Lab, and Professor Karthik Ramachandran, Professor Raghupathy Sivakumar, and Rahul Saxena from Create-X at Georgia Tech for their help, invaluable advise, and encouragement in my entrepreneurial pursuits.

I would like to thank the wonderful College of Computing Staff Dani, Kevin, Trinh, Carlos, Tiffany, and Jonathan for their help throughout.

I would like to thank my parents and sister for their love and support to pursue my

dreams. Special thanks to Anveer, my nephew for keeping my sanity in check during this journey. I would like to thank my wife for providing the inspiration and being the rock-solid support through the thick and thin.

Last but not least, I am thankful to my friends, Devdutt, Vinayak, Milind, Jai, Devesh, Ram, Satish, KD, Raju, Imran, and Sheetal for their support throughout.

I look forward to starting the next chapter in my life with my soon-to-be-born son!

TABLE OF CONTENTS

Acknowledgments iv

List of Tables x

List of Figures xii

Summary xiv

Chapter 1: Introduction 1

1.1 Problem Statement 1

1.2 Thesis Overview 3

1.2.1 Modern mobile apps 3

1.2.2 Context-sensitive storage management 5

1.2.3 Extensible user file system framework 6

1.3 Thesis Contributions 7

Chapter 2: Background 9

2.1 Android app anatomy 9

2.2 Android app storage areas and partitions 10

2.3 Android app storage directories 11

2.4 Android app installation process 12

2.5 Android app storage abstractions 13

Chapter 3: Large-scale analysis of mobile storage consumption 14

3.1 Android app storage consumption 14

3.2 Modern Mobile Apps 15

3.2.1 Research questions and methodology 15

3.2.2 Findings 16

3.3 Static Analysis 22

3.3.1 Research questions and methodology 22

3.3.2 Findings 23

3.4 Install-time behavior. 30

3.4.1 Research questions and methodology 31

3.4.2 Findings 31

3.5 User study 32

3.5.1 Research questions and methodology 33

3.5.2 Findings. 35

3.6 Mobile storage management 42

3.6.1 Design space: challenges and opportunities 43

Chapter 4: Extensible User File system for Storage Management 47

4.1 Overview 47

4.2 eBPF 49

4.3 Architecture 51

4.4 Workflow 52

4.5 APIs and Abstractions 54
4.6 Implementation 55
4.7 Optimizations 56
4.7.1 Customized in-kernel metadata caching 56
4.7.2 Passthrough I/O for stacking functionality 59
4.8 Evaluation 60
4.8.1 Performance 62
4.8.2 Use cases 66

Chapter 5: Context-aware storage architecture 71

5.1 Use case 71
5.2 Goals 72
5.3 Challenges 72
5.4 Storage Reclamation 73
5.5 Continuous System Profiling 77
5.6 Architecture and Workflow 78
5.7 Context-sensitive Storage Profiles 79
5.8 Storage Management 83
5.9 Implementation 84
5.9.1 File System 85
5.9.2 Storage service 85
5.9.3 Continuous monitoring. 86
5.10 Evaluation 87

5.10.1 Micro benchmarks . . . 87
5.10.2 Storage Savings . . . 90
5.10.3 Prediction Accuracy . . . 90

Chapter 6: Discussion . . . 93

Chapter 7: Related Work . . . 95

7.1 Network File Systems . . . 95
7.2 Distributed File Systems . . . 95
7.3 Other Relevant Storage Systems . . . 96
7.4 Data Replication Systems . . . 97
7.5 Data Adaptation . . . 97
7.6 Storage Management . . . 98
7.7 Context-aware Computing and Prefetching . . . 99
7.8 Cloud Storage Services . . . 99
7.9 User File System Frameworks . . . 100
7.10 Extensible Systems . . . 101
7.11 Study of Mobile apps stores . . . 102

Chapter 8: Conclusion . . . 103

References . . . 117

LIST OF TABLES

2.1 Supported CPU architectures under Android. 10

2.2 Storage paritions on Android devices. 10

2.3 App storage areas on Android devices. 12

3.1 App storage consumption phases. 15

3.2 Increase in the number and average sizes of apps across categories. 19

3.3 Increase in native libraries supporting multiple CPU architectures. 27

3.4 Evolution of top Android apps over time. 29

3.5 Summary of contextual data collected from our user study. 35

3.6 Summary of device events we subscribe to for data collection from our user study. 35

3.7 Summary of file system APIs hooked for data collection from our user study. . . 36

3.8 Snapshot of storage consumption on devices across 21 users 40

4.1 EXTFUSE APIs and abstractions. 54

4.2 EXTFUSE kernel changes. 56

4.3 Metadata caching with EXTFUSE. 56

4.4 Different StackFS configs evaluated. 63

4.5 EXTFUSE adoption effort. 67

4.6 EXTFUSE evaluation with Android FUSE daemon. 69

5.1 Storage reclaimation techniques used by ANODYNE. 75

5.2 Contextual data collected. 82

5.3 Evaluation profiles. 87

LIST OF FIGURES

3.1 Increase in app sizes over the years. 17

3.2 App store submission policy changes over time. 17

3.3 Distribution of apps across categories. 18

3.4 Comparison of app downloads across app sizes. 20

3.5 Longitudinal analysis of app sizes vs. app downloads. 20

3.6 Static analysis of apps. 24

3.7 Increase in the number of features over time in top apps. 25

3.8 Analysis of native libraries found across 1.1 and 1.7 million free apps. 26

3.9 Install-time storage consumption analysis of top apps. 32

3.10 Breakdown of `/data` storage partition of users. 37

3.11 Breakdown of storage consumption of apps. 37

3.12 Free space available on the device for user-O over time. 38

3.13 Fitbit app storage consumption over time for user-O. 39

3.14 Growth in mobile storage capacity. 42

3.15 OBB accesses across three different levels 46

4.1 ExtFUSE architecture. 51

4.2 Changes to FUSE daemon `lookup` handler under ExtFUSE. 57

4.3 EXTFUSE `lookup` kernel extension that serves cached replies. 58

4.4 Changes to FUSE daemon `open` handler under EXTFUSE. 61

4.5 EXTFUSE `read` kernel extension that enables passthrough access. 61

4.6 Throughput (ops/sec) results for EXT4 and FUSE/EXTFUSE Stackfs file systems as measured by data micro-workloads. 62

4.7 Number of file system requests received by the daemon in FUSE/EXTFUSE Stackfs. 62

4.8 Throughput (ops/sec) results for EXT4 and FUSE/EXTFUSE Stackfs file systems as measured by metadata micro-workloads. 63

4.9 Linux kernel compilation evaluation under FUSE and EXTFUSE. 67

4.10 Android `sdcard` permission checks EXTFUSE code. 69

4.11 LoggedFS kernel extension that logs requests. 70

4.12 LoggedFS performance evaluation under EXTFUSE. 70

5.1 ANODYNE Mobile Client Storage Service Architecture. 79

5.2 Storage prediction algorithm and workflow. 79

5.3 An excerpt from user data traces. 81

5.4 Kernel extension that traces I/O reqs. 86

5.5 Memory and battery overhead from top thirty apps. 89

5.6 CPU and latency overhead from top thirty apps. 89

5.7 Storage and network overhead from top thirty apps. 89

5.8 Per-day prediction accuracy. 91

5.9 Per-hour prediction accuracy. 92

SUMMARY

Smart mobile devices have largely evolved as primary tools for personal computing needs. There are millions of applications (or apps) for everyday tasks, such as social networking, entertainment, healthcare, and home automation. There is an increasing trend of using personal devices for work as well. Given the limited storage capacity, users quickly find the device running out of storage space. Existing cloud storage services augment storage on mobile, but they require manual data management. Therefore, efficient management of limited storage on these devices is becoming increasingly important. Yet, no systematic study has been conducted to analyze mobile storage utilization by modern apps. This work aims to study and address storage management challenges on smart mobile devices.

We present the first ever large-scale and detailed measurement analysis of growing storage footprint of apps on smart mobile devices. We begin by carrying out a longitudinal study of millions of Android apps to report the increase in their installation sizes over five years. Our study reveals that modern apps have evolved as large installation packages that pose high storage demands (over 4 GB). Our comparative analysis of installation sizes of millions of modern iOS apps reveals similar findings.

We further perform static code analysis of apps to report various sources of install-time storage consumption. We found that modern apps consist of not only proprietary binaries, but also third-party libraries (e.g., social media plugins) and various auxiliary files (e.g., animations, icons) to provide rich user experience. Furthermore, as apps become popular, developers pack more (up to 2x) features to engage users and monetize, resulting in big monolithic apps. Finally, despite the push from vendors to create small device-specific apps, developers create "build once, run everywhere" universal apps to target multiple hardware architecture types and demographics that consume redundant storage.

We then carry out a user study with hundreds of Android participants across various age groups, demographics, and professions to analyze post-installation storage utilization

behavior of mobile apps. Our findings suggest that today's monolithic apps are not optimized for storage. Upon installation, app binaries are decompressed and optimized for performance. A typical user has hundreds of such performance-optimized apps on their device that in total consume a sizable amount of device storage. However, our storage traces reveal that users actively interact with only 10% of apps and installed features, on average, and the usage is highly correlated with the user context (e.g., day, time). Furthermore, we found that apps are not constrained by storage quota limits; developers freely abuse persistent storage by frequently creating debug logs, user analytics data, caching Cloud content for native performance, and advertisements for monetization as needed.

Drawing upon our study findings, we then propose a context-sensitive quota model for automatic storage management on smart mobile devices. We present the design and implementation of our prototype platform storage for Android that performs context-aware proactive storage management. The mechanisms introduced, however, are generic and apply to all modern mobile operating systems. Storage space consumed by contextually unwanted apps/data is transparently reclaimed by employing multiple techniques, such as compression, deletion, content adaptation, deduplication, and cloud-backed hierarchical management. Reclaimed data is reconstructed proactively under predictive usage or served on-demand either by computation on the device or fetching it from the cloud. We design a novel file system framework to stack extensible storage management functionality layer as a file system in the user space. This offers app-transparency and compatibility with existing storage interfaces.

CHAPTER 1
INTRODUCTION

1.1 Problem Statement

Smart mobile devices have largely evolved as primary tools for everyday personal computing needs, including communication, travel and planning, gaming, health tracking, social networking, home automation, media, and entertainment. Personal mobile devices are also increasingly being used for work. The versatility of these devices poses high, often conflicting, storage demands.

Storage-heavy apps. With millions of applications (or apps for short) available at their disposal [1], users often download and install them for customized experience. Historically small in size (up to 25 MB [2]), modern mobile apps are large and complex installation packages that pose heavy storage demand. The maximum permissible app sizes of Google Play Store Android and Apple App Store iOS apps have only been growing since 2008 as shown in Figure 3.2. Today, installation of a single app can consume over 4 GB of storage space [3, 4]. While anecdotal evidence suggests that user data (e.g., pictures, videos) consume high storage space, no systematic study has been carried out about the storage consumption behavior of modern apps.

Limited storage. The problem of high storage demand is exacerbated by limited storage capacity of smart devices. Many of them are not provisioned with a removable external flash memory [5]. Low-end budget devices with limited storage capacity, performance, and low price are still prevalent, especially in developing countries. Such devices place further restrictions on storage, severely limiting the user experience [6]. Moreover, a high percentage of available storage is consumed by the operating system (OS) resources [7] and pre-installed *bloatware* from vendors and carrier providers [8]. For instance, 16 GB

Samsung Galaxy S4 comes with only 8.56 GB of free space out of the box [9]. The remaining free space is, however, shared among all installed apps, thereby hosting app executable content (e.g., binaries, libraries, etc.), app private data (e.g., user preferences, account info, favorites, game progress, etc.) created during usage, cached content (e.g., web or cloud data, temporary files, etc.), and any user created data (e.g., documents, photos, movies, music, books, etc.). Yet, installed apps are not constrained by storage consumption limits. A single app can potentially consume up to 90% of the storage on Android [10].

Existing workarounds. Through the daily and extensive use, mobile devices accumulate large amounts of data and quickly exhaust storage space. As a workaround, users are forced to temporarily uninstall apps, backing up, or deleting data [11]. This introduces two serious problems. First, despite the availability of millions of apps, only a few could be used at a time [12], compounding the issue of customer retention that developers are already struggling with in today's immensely crowded app market. Second, users have to manually manage storage by backing up, deleting data from the device to reclaim storage space, and downloading it again as needed. However, manually sifting through the files to decide what to safely delete or store for everyday use can not only quickly become onerous, but also lead to errors and wastage of storage.

Existing personal Cloud storage services such as Dropbox[13], Box [14], and Google Drive [15] allow easy backup, sharing, and data synchronization across devices. However, none of them provide automatic storage management, and merely shift the problem to wasting the Cloud storage. Users still have to manually upload data that needs to be backed up or shared and delete from the device to reclaim space. iCloud [16] service that is integrated with iOS, simply offloads infrequently used app executables to Cloud [17], but offers no way to remove app data without uninstalling the app [18]. Similarly, cleaner apps either delete valuable user data along with the app or need superuser privileges to circumvent app sandboxing on Android for full functionality. Alternatively, users regularly upgrade their devices by paying a premium price for additional mobile storage capacity [19].

1.2 Thesis Overview

This work aims to addresses storage management challenges on smart mobile devices. We present the first large-scale and detailed measurement study of storage utilization behavior of mobile apps. Based on our study findings, we then present an automated storage management architecture for mobile devices. The following sections introduce our work.

1.2.1 Modern mobile apps

App store study. We carried out a large-scale longitudinal study of millions of Google Play Store Android apps to report increase in their installation-time storage footprint over five years, from 2014 to 2019 §3.2. Our results show that modern mobile apps have evolved as large monolithic installation packages. As app stores increased app size limits over time, developers created feature-rich apps that pose heavy storage demands. The number of apps consuming between 10 MB to 4 GB doubled in five years. Such apps are not limited to games and digital books, but spread across various categories.

We also analyzed metadata of millions of iOS apps, and report their storage requirements for comparative analysis. The average size of an iOS app is 3.5x that of an Android app.

App Static analysis. We further performed longitudinal static code analysis of millions of Android apps, and highlight various sources of growth in app sizes over time §3.3. Our findings show that as an app gains popularity, developers pack more features in the same app, creating *super apps* to engage users and monetize. Number of features in top apps doubled in five years. While paid features are only unlocked once the user purchases them, they are needlessly always stored on the device. The number of third-party libraries imported per-app for common tasks such as user authentication and advertisements grew by 10x, adding to app size.

Furthermore, we found that developers continue to create "build once, run everywhere" *universal* apps, despite Google's effort to encourage developers to create small device-

optimized apps [20, 21]. Such apps not only support multiple hardware architectures (e.g., x86, MIPS), but also various regional languages to cater to diverse demographics, trading more storage.

Install-time analysis We also evaluate storage consumption behavior of modern apps during fresh installation (no usage) on mobile devices by analyzing top 30 Google Play Store apps, each with 5 million downloads. We found that upon installation, apps further expand to consume at anywhere between 1.5x to 5x storage space of their installation size §3.4. This increase is attributed to additional files created for performance.

Post-install user study Finally, we report runtime storage behavior of Android apps by leveraging file system traces from our user study with over 140 participants for 70 days. We built COSMOS, a novel lightweight context-aware storage tracing tool, and deployed it on each participant's device.

We found that each user has 122 apps installed on their mobile device that consume almost 50% of the total storage, on average. However, the traces we collected not only revealed that a small number of app features (and files) are actively used, but also suggests that storage usage is highly correlated to user context.

Analyzing the storage traces, we further found that apps are not constrained by storage quota. App developers freely use persistent storage by frequently caching and hoarding data as needed to offer high performance and rich user experience. Many apps hoard additional data, such as analytics to track user engagement, crash reports for debugging, and advertisements for monetization. However, unlike cached content, which is automatically deleted when the device runs low on storage space, hoarded app data continues to persist on device, even after the apps are uninstalled. We detected several old video and image advertisements and several residual multimedia/text data files on users devices.

1.2.2 Context-sensitive storage management

One way to manage mobile storage usage is to enforce per-app quota to limit consumption. Nonetheless, traditional fixed-size quotas assume that all apps are equally important to the user and can lead to wastage of storage if apps use up only a small fraction of their quotas or not scale as richer functionality is introduced and apps grow in size. Elastic quota model [22] overcomes the aforementioned shortcomings by hard-limiting only persistent data and allowing temporary data to grow or be reclaimed depending upon the available storage space. However, it requires the user to identify what data should persist or thrown away and when, ruling out the possibility of automated and application-transparent storage management. This is because the relative importance of apps not only varies with the user (e.g., gamer vs. non-gamer), but also varies for the same user depending on the context (e.g., home vs. office) as shown by our user study. Furthermore, imposing per-app hard limits require redesigning of apps to work with limited quota and provide the same functionality with no discernible impact on user experience.

In the light of above findings, this work proposes a *context-sensitive storage quota model* for automated storage management of smart devices. We assert that since smart devices are personal consumer devices, *modern operating systems must proactively manage the storage resources on behalf of the user depending on storage requirements for a specific user context*. Smart devices are equipped with a number of sensors and present a plethora of opportunities to build active user context. In fact, many mobile operating systems (e.g., Android, iOS) already host a personal assistant that tracks user's habits and provide context-based proactive recommendations on next apps to use. We propose to extend the personal assistant on the device to also learn about user's everyday storage needs and automatically build working set profiles for multiple contexts, such as location, day/time, and calendar. Based on the profiles, the system can then transparently enforce quota-driven consumption or reclamation of space for more productive use of local storage. The key observation we make is that users typically use only a fraction of installed apps actively, and the usage follow a fixed,

repetitive schedule as shown by our user study. For example, if particular set of music files are always accessed every morning, if work-related data is only accessed during weekdays or at a particular location, or if personal photos and videos are less likely to be accessed at work.

To understand performance and usage implications of our proposed design, we have prototyped a context-sensitive automated storage management architecture called ANODYNE for Android mobile devices. We present its design and implementation details in this work. Based on user context, ANODYNE either allows unrestricted use of available storage (e.g., during active or predictive use of apps) or automatically reclaims space when low on storage by employing multiple storage management techniques, such as compression, deletion, content adaptation, deduplication, and Cloud-backed hierarchical management. Reclaimed data is reconstructed proactively under predicted usage or served on-demand either by computation on the device (e.g., decompression) or fetching it from the Cloud. ANODYNE also provides optional APIs to apps for enforcing custom storage management policies, and backup data to their Cloud servers before reclamation.

1.2.3 Extensible user file system framework

We have designed ANODYNE to particularly minimize performance overhead, efficiently trade budgeted resources (e.g., battery) for storage, and demonstrate practical feasibility of our approach. For instance, we use content-aware offline deduplication to reclaim space occupied by duplicate I/O blocks in app executables. To enable automated and app-transparent storage management, ANODYNE poses itself as a file system. This also offers compatibility with existing storage APIs. We developed an extensible user file system framework, called EXTFUSE that allows us to implement our file system in user space and support multiple complex storage management techniques (e.g., deduplication, compression) in the user space. Being extensible, EXTFUSE allows us to further extend the kernel at runtime by registering specialized handlers for serving a low-level file system

requests in the kernel, without always context switching to the user space. As such, it offers performance of kernel file systems, while retaining our existing complex storage management functionality in user space to achieve the desired level of performance. We tap into EXTFUSE to stack lightweight continuous I/O tracing, read de-duplicated blocks, and directly forward data I/O requests through the underlying file system in the kernel at native speed if no data reconstruction is required. Whereas, complex storage management tasks (e.g., on-the-fly decompression) are handled in the user space for better system reliability. As such, ANODYNE always offers native performance for apps that are predicted to be used.

1.3 Thesis Contributions

In summary, this thesis makes the following contributions to storage management on smart mobile devices.

- **Large-scale app analysis.** We carry out the first large-scale longitudinal study of millions of Google Play Store Android and Apple App Store iOS apps to highlight increase in their install-time storage footprint over time §3.2. We further perform static analysis of apps to report various sources of storage consumption §3.3.
- **User study.** We carry out a user study with 140 Android users in the wild on post-installation storage consumption behavior of modern mobile apps §3.5. Leveraging collected file system traces and app usage stats, we present several detailed insights into the landscape of mobile storage consumption by modern apps §3.6. We believe our analysis will be a highly valuable to app developers as well as the storage community for improving mobile storage management.
- **Context-sensitive storage quota.** Drawing upon our study findings, we introduce the idea of context-sensitive quota and propose a framework, called ANODYNE, for automated context-aware storage management of smart devices §5.
- **Extensible Userspace File System.** We present EXTFUSE, an extension framework for user file systems that offers the performance of kernel file systems, while retaining

the safety properties of user file systems §4.

- **Prototype implementation** We present the design and implementation of our prototype storage system service around ANODYNE on Android, and evaluate its performance §5.6.

CHAPTER 2
BACKGROUND

This work focuses primarily on Android mobile ecosystem because of its popularity and market dominance. In this section, we cover basics of Android app anatomy as well as present a detailed background on different storage areas available to mobile app developers on Android.

2.1 Android app anatomy

Android support two types of development environment, namely Java and native C/C++. A Java source file is first complied into a bytecode `.class` file, which is then linked with required third-party `.jar` libraries to produce a `classes.dex` file. Each DEX file can contain up to 65,000 methods. Therefore, large apps typically consist of many `.class` files and split functionality across multiple DEX files. In contrast, C/C++ sources files are compiled into dynamic `.so` libraries that contain native or machine (e.g., ARM, x86, etc.) code. Android supports seven different types of machine architectures as listed in Table 2.1. Google encourages developers to create multiple apps, each optimized for specific architecture [21].

All DEX files, along with native libraries and auxiliary *asset* files (e.g., icons, images) are packed into a single App Package (APK) zip archive. When an app is installed on a device, its APK is downloaded from Google Play Store and all executable as well as auxiliary resource files are extracted from APK and stored on the device for faster access.

In addition to the APK file, developers can optionally provide up to two monolithic *expansion files* [24] in the form of Opaque Binary Blobs (OBB) to add any auxiliary resources (e.g., images, videos) as needed. OBB files supplement the APK without bloating the main executable and enable more complex apps, such as graphics-rich 3D games.

Table 2.1: **Supported CPU architectures under Android.** Different types of CPU Instruction Set Architectures (ISA) and Application Binary Interfaces (ABI) supported by Android [23].

Machine Family	CPU Architecture	ABI
ARM	ARMv5, 32-bit	armeabi
ARM	ARMv7, 32-bit	armeabi-v7a
ARM	ARMv8, 64-bit	arm64-v8a
x86	x86, 32-bit	x86
x86	x86_64, 64-bit	x86_64
MIPS	MIPS, 32-bits	mips
MIPS	MIPS64, 64-bits	mips64

Table 2.2: **Storage paritions on Android devices.** A large part of *internal* `/data` parition is exposed as *primary external* `/data/media` partition as a FUSE managed file system. Besides these, there are other partitions on Android devices, such as `/vendor`, `/oem`, and `/odm` [25].

Partition	Type	FS	Examples
/	Internal	RootFS	Bootup and init scripts (e.g., init.rc)
/system	Internal	EXT4	System libs, utils, and apps (e.g., email)
/data	Internal	EXT4	User apps, private app data (e.g., databases)
/data/media	Prim Ext	FUSE	Public app data (e.g., OBB), user data (e.g., pics)

2.2 Android app storage areas and partitions

Android offers two types of storage partitions, namely `internal` and `external`. The internal storage is a built-in non-volatile flash memory containing critical system partitions, such as boot (for bootloader and kernel), recovery, system (containing system software), and data (hosting apps). It is, however, not directly accessible to the user and managed by EXT4 file system [26].

In contrast, external storage contains only of data partition and is directly available to user (e.g., USB plugging). A device can contain two types of external storage areas – non-removable (called *primary*) and removable (called *secondary*) external storage areas. An example of a secondary external storage area would be the `/sdcard` partition that is mounted when an external removable flash memory card is plugged into the device. However, many smartphones are not provisioned with a removable external storage [5]. On such devices,

a part of the built-in internal storage is typically exported as external. Therefore, in this work, we only focus on the storage consumption of internal and primary external storage areas since they are fixed in size (non-removable) and the storage space shared by all apps with no per-app quota limits. Android starting from version 4.3 [27] rely on FUSE user file system [28] to manage the primary external storage and enable multi-user support.

2.3 Android app storage directories

When an app is installed, the Android system creates app-specific directories in both default internal (under `/data/data/`) and an external storage (`/sdcard/Android/data/`) data directories. These directories are named after the app package name. Table 2.3 shows various designated app data directories on Android. Directories in internal storage are private and exclusively owned by respective apps. Whereas, directories in external storage are public and can be read and written by apps with `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` permissions respectively. Apps with the read permissions to external storage can access everything stored on there, even data from other apps. For Android versions before 4.4, external storage is world-readable by default and requires no permissions.

App private directories are primarily intended for hosting code (e.g., java classes, native libraries, etc.) and private data (e.g., user preferences, account info, game progress, etc.). Whereas, public directories are typically used for storing large app auxiliary data (e.g., game OBB files) and any user content such as music, videos, photos, and documents Table 2.3. App directories are further divided based on type of content stored so that developers can pick the appropriate storage options. For example, it is advisable to store temporary data, such as web content and temporary files under `cache` because the caches are cleared by the system when the device runs low on storage space.

Table 2.3: **App storage areas on Android devices.** Each app is assigned private (under internal) as well as public (under primary external) storage areas (dirs) to host its code and data. App files residing in public app storage area can be accesses by all other installed apps with read/write permissions. Typically app code (java classes and native libraries) and structured data (e.g., databases, etc.) are stored in private app dirs and large unstructured files (e.g., OBB files) are stored in public app dirs. APP APK files and their performance-optimized OAT files are stored in internal data partition.

App Storage Location	Type
/data/app/app/base.apk	APK file
/data/dalvik-cache/app	Pre-JITed OAT file
/data/data/app	Private app dir*
/data/media/Android/data/app	Public app dir*

App Dir*	Type
app/files/	Java Code, Data
app/libs/	Native Code
app/databases/	State (SQLite)
app/shared_prefs/	State (XML)
app/cache/	Cached Data

2.4 Android app installation process

During app installation, all binary files (DEX and libraries) are extracted from the APK and separately stored on the device, under app-specific directories in the internal storage partition. On the other hand, OBB files, being large in size, are stored on the external partition under an app-private directory.

Unlike native libraries, the DEX bytecode format is independent of machine architecture and needs to be compiled to native machine code to run on the device. Older Android versions used Dalvik Virtual Machine (VM) runtime that performed Just-in-Time (JIT) compilation of DEX files. Version 5 introduced Ahead-of-Time (AOT) compilation of DEX files into optimized `OAT` files during installation. AOT compilation of Java bytecode is configurable. It ranges from compile "everything" to "interpret only". While the former improves app runtime performance, the latter provides performance similar to Dalvik VM. By default, most of the methods in an OAT file are precompiled to maximize runtime performance. As a result, OAT files consume significant storage space and incurs a longer installation time.

2.5 Android app storage abstractions

Android provides various storage abstractions, such as files, databases, and xml schema. App developers can choose to store raw bytes in the form of traditional files or make use high-level storage abstractions such as databases and xml schema to store formatted data. Based on type of kind of content (raw bytes or formatted data), developers can pick the appropriate app directory to store it. For example, it is advisable to store temporary data such as web content (e.g., news feeds) and temporary files under `cache` because when the device runs low on storage space these temporary files are automatically deleted and the storage space occupied app caches is reclaimed by the system. In contrast, `databases` and `shared_prefs` directories contain Stateful persistent data (e.g., user preferences, account info). Such persistent data is critical to user experience; thus, cannot not be simply deleted to reclaim storage.

CHAPTER 3

LARGE-SCALE ANALYSIS OF MOBILE STORAGE CONSUMPTION

Understanding how modern mobile apps use available storage resources is crucial to designing an efficient storage management system. Therefore, we carried out a large-scale measurement analysis of storage consumption behavior of modern apps on smart mobile devices. Here we describe our methodology, and reports our findings.

3.1 Android app storage consumption

We categorize the storage consumption of mobile apps into three phases depending on when it occurs as shown in Table 3.1. Pre-install phase is the first step immediately upon receiving a fresh installation request for an app from the user. During this phase, the consumption refers to the storage space occupied by app installation files such as Android APK and OBB files are downloaded from the app store to the device. Pre-install consumption results from app installation files downloaded from the official app store during a fresh installation.

Install-time consumption refers to the storage space occupied by files created during app installation, such as pre-compiled OAT and unzipped native libraries. Some apps may download additional files from a remote server during installation. Such files also add to the install-time consumption. Usage-time consumption occurs as a result of files downloaded from a Cloud server (e.g., Ads) or created locally on the device (e.g., crash/debug logs).

We performed three separate studies to analyze the storage consumption behavior of Android apps in each of the aforementioned phases, namely pre-install, during installation, and runtime usage, respectively.

Table 3.1: **App storage consumption phases.** We categorize the storage consumption of apps into three phases. Pre-install consumption results from app installation files downloaded from the store when an installation request is received. Install- and usage-time consumption occur as a result of files created during fresh installation and active usage of an app, respectively.

Phase	Sources of storage consumption
Pre-install	Files downloaded (e.g., APK, OBB) from the offical app store.
Install-time	Files downloaded from a Cloud server or created (e.g., OAT, libs).
Usage-time	Files downloaded (e.g., Ads) or created (e.g., logs) during usage.

3.2 Modern Mobile Apps

To understand how mobile apps have evolved over time, we performed a large-scale longitudinal study of millions of Google Play Store Android apps from the last five years. To provide a comparative analysis of app sizes, we also performed a measurement study of iOS. In this section, we report our study methodology and findings.

3.2.1 Research questions and methodology

Our study answers the following questions.

App sizes. *How much storage is consumed by app installation files? How have app sizes changed over time?*

App size vs. device. *Does the user device type (tablet vs. phone) or config (high- vs. low-resolution display) affect app installation size?*

App size vs. category. *What effect does the app category have on its installation size? What categories are likely to be large or small in size?*

App size vs. downloads. *What effect does the app size have on its popularity and vice-versa (e.g., are large apps more popular than small ones?)*

Methodology. Popular mobile app stores such as Google Play Store and Apple App Store provide APIs to fetch the most current metadata on all apps, such as app name, version, category, release date, developer information, number of downloads, and app download size.

Unlike monolithic iOS apps, the size of an Android app is the sum of its APK size and sizes of all auxiliary OBB files that the app downloads during installation. We wrote scripts to automate the process of downloading and analyzing app metadata. Specifically, we looked at the download size of each app, the category it falls under, and its popularity (i.e., number of downloads). It is worthy to note that Google Play Store does not provide a precise number of downloads or installs per app; it only provides download range approximations. Apple App Store, on the other hand, provides no download statistics.

In this study, we only consider the pre-install storage footprint of apps; i.e., storage space consumed by all app installation components downloaded from the official app store (see Table 3.1). Installation-time storage footprint statistics are provided in §3.4.

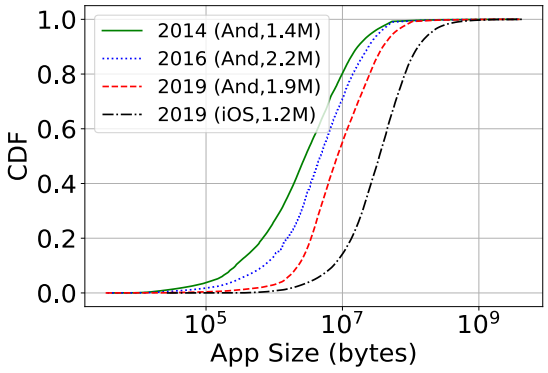
Dataset. Our dataset consists of metadata (e.g., name, size, developer name, etc.) of millions (M) of Google Play Store Android and iOS apps. It includes,

- 1.4M Android apps collected in October, 2014 by PlayDrone [29] project.
- 2.2M Android apps we collected in December, 2016 for our OSSPolice [30] project.
- 1.1 million and 1.9 million apps that we collected for this study in July, 2018 and August, 2019, respectively. Our Google Play Store crawler was based on [29].
- 1.2 million iOS apps collected from App Store in August, 2019.

3.2.2 Findings

App sizes. Figure 3.1a shows our findings on app installation sizes. We found a sharp increase in both the number of available Google Play Store Android apps and the average app installation size (7.89 MB in 2014 to 17.16 MB in 2019) in five years. Only 20% of apps were more than 10 MB in size in 2014. That number grew by 50% in 2016 and doubled in 2019; that is, over 40% of the apps in 2019 consumed over 10 MB in size. We also found a small increase in the number of large apps ($>$ 1 GB in size) from 2014 to 2019.

Furthermore, over 5% apps in 2019 consumed up to 100 MB in size, compared to only 0.3% such apps in 2014 (see Figure 3.1b). We believe this is due to the change in Google



(a)

Max Size	Num Android Apps (%) *2014 (1.4M)*	Num Android Apps (%) *2019 (1.9M)*	iOS Apps (%) *2019 (1.2M)*
50MB	99.224	94.139	67.60
100MB	0.3221	5.0365	19.72
1 GB	0.4395	0.7973	12.46
2 GB	0.0124	0.0243	0.177
3 GB	0.0008	0.0016	0.027
4 GB	0.0001	0.0006	0.007

(b)

Figure 3.1: Analysis of app installation sizes of millions of Google Play Store Android and iOS apps from 2014-2019. Our analysis shows that both the number and average app size have been increasing over last five years.

Play Store app submission policies in 2015 that allow developers to publish an app with APK size of up to 100 MB, which doubled the prior limit of 50 MB [31].

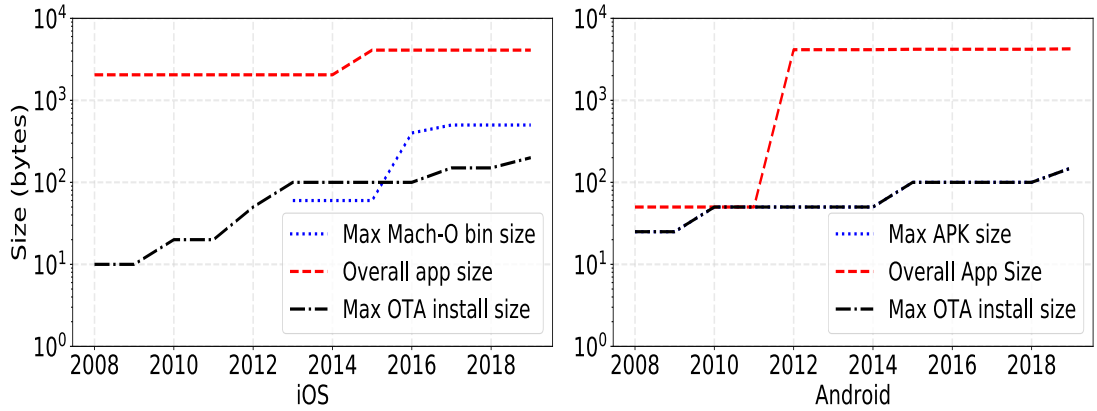


Figure 3.2: **Changes made to Google Play Store Android and Apple App Store iOS app submission policies over time [2, 3, 4].** Both Google Play Store and Apple App Store have been increasing maximum permissible app sizes as well as Over The Air (OTA) download limits to allow developers to pack more features in apps.

Similar increase in maximum permissible app size was made by Apple for iOS apps [4]. As a result, we found that 32.4% of 1.2 million iOS apps we analyzed consumed more than 50 MB. 19.72% and 12.46% of apps were $>$ 100 MB and 1 GB, respectively.

Additionally, our findings suggest that iOS counterparts of Android mobile apps are

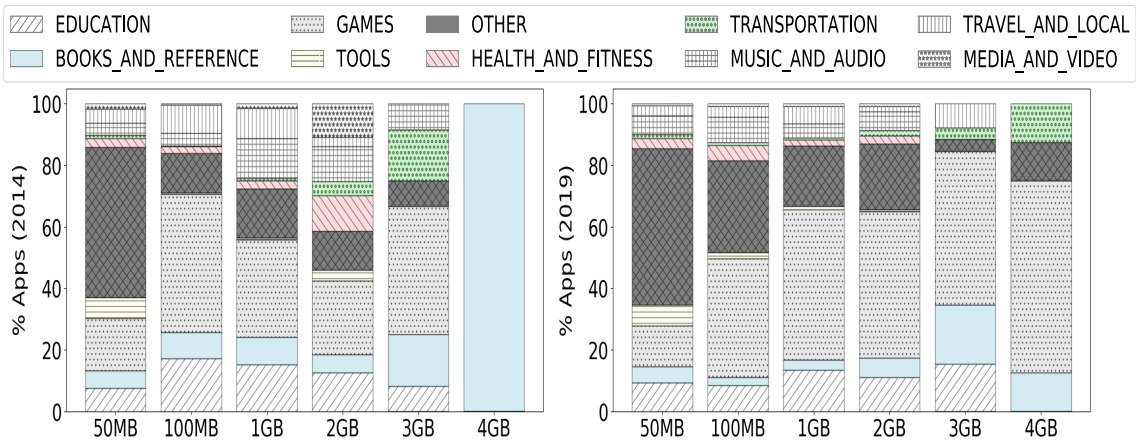


Figure 3.3: **Distribution of app sizes across categories.** Millions of Google Play Store Android apps collected in 2014 and 2019 show that apps larger than 1 GB in size are not only limited to gaming or digital book apps, but spread across multiple categories, such as education, tools, music and video. Furthermore, the number of such large apps, particularly gaming apps increased substantially from 2014 to 2019.

---

larger in size. For example, Facebook and Uber iOS apps are 410 MB (20%) and 311.6 MB (32%) larger than their Android versions, respectively. Overall, we average iOS app size to be 56.65 MB, which is 3.2 times the average Android app size in 2019.

App size vs. category. We compared apps from 2014 and 2019 across different categories on their installation sizes. Contrary to common intuition, we found that apps larger than 100 MB (i.e., max APK size) is not limited to graphics-rich games or digital books, but in fact are spread across various categories, including music/video, education, and travel apps. These apps make use of OBB expansion files to include supplemental data needed by the app, while keeping the main executable or APK separate.

Figure 3.3 shows the percentage distribution of apps categories against app installation sizes. The only apps larger than 3 GB in 2014 were digital book apps, such as SiKu QuanShu and Wiki Encyclopedia. In 2019, however, 62.5% of such apps ($>$ 3 GB) were games, and only 12.5% were books. We also saw increase in size of transportation and navigation apps.

We also found that the average sizes of apps across categories grew substantially in five years. Table 3.2 shows our results. For instance, the average size of a gaming app increased

Table 3.2: **Increase in the number and average sizes of apps across categories.** The average sizes of both Google Play Store Android and iOS apps grew substantially from 2014 to 2019 across all categories. The average size of a gaming app increased almost 2.5x.

App Category	Android (2014, 1.4M) Num Apps (%)	Android (2014, 1.4M) Avg Size (MB)	Android (2019, 1.9M) Num Apps (%)	Android (2019, 1.9M) Avg Size (MB)	iOS (2019, 1.2M) Num Apps (%)	iOS (2019, 1.2M) Avg Size (MB)
GAMES	17.26	13.73	14.90	33.72	16.80	103.86
EDUCATION	7.76	10.29	9.34	16.74	10.12	71.19
TRANSPORT/MAPS	1.23	5.16	1.24	14.06	1.27	62.59
TRAVEL/LOCAL	4.66	9.33	3.22	19.27	4.44	67.49
BOOKS/REFS	5.56	8.44	5.02	12.77	2.94	70.75
MUSIC/AUDIO	3.83	10.58	6.20	15.76	2.91	51.32
MEDIA/VIDEO	1.68	7.02	0.64	19.70	2.37	48.80
HEALTH/FITNESS	2.85	7.12	3.40	18.26	4.29	55.20
TOOLS/UTILS	6.61	2.30	6.44	8.15	6.58	35.64
OTHER	48.56	4.92	49.60	13.22	48.28	45.34
OVERALL	100	7.89	100	17.16	100	61.22

almost 2.5x from 13.73 MB in 2014 to 33.72 MB in 2019. Similarly, the average size of digital books apps increased from 8.44 MB in 2014 to 12.77 MB in 2019.

App size vs. downloads. Figure 3.4 shows the cumulative distribution of apps downloads (installs) against their installation size. We use the download range to understand popularity of Android apps. We found that large apps more popular than small apps. In 2014, 20% of apps more than 50 MB in size were downloaded at least 1000 times, whereas 30% of apps between 50 MB and 100 MB received that many downloads Similarly, 47% of apps more than 3 GB in size received more downloads in 2019 compared to apps smaller than 2 GB. Even though Android warns the user when installing apps larger in size than the permissible Over The Air (OAT) limits, which was 50 MB in 2014 and increased to 100 MB in 2015 Figure 3.2, we found that overall a higher percent of such apps have been downloaded at least million times in both 2014 and 2019.

Top apps. Our aforementioned findings on app download behavior suggest that the bigger the app, the more number of downloads it is likely to receive. To further understand the reverse affect, we looked at the average increase in size of apps across various download ranges. We found that popular apps grew more in size on average over the years compared

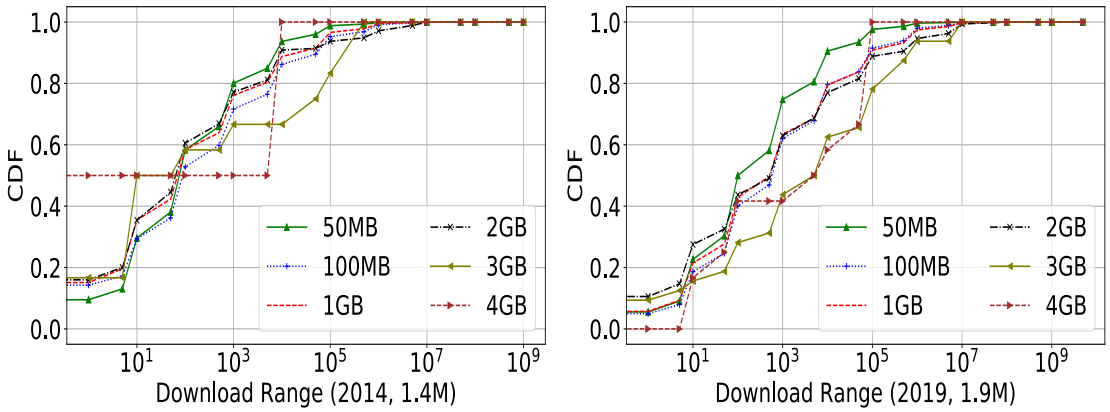


Figure 3.4: Comparison of downloads (installs) across app sizes in 2014 and 2019 for millions of Google Play Store Android apps collected in 2014 and 2019. Larger apps (>1 GB in size) are more popular than smaller apps, and this behavior is consistent across time.

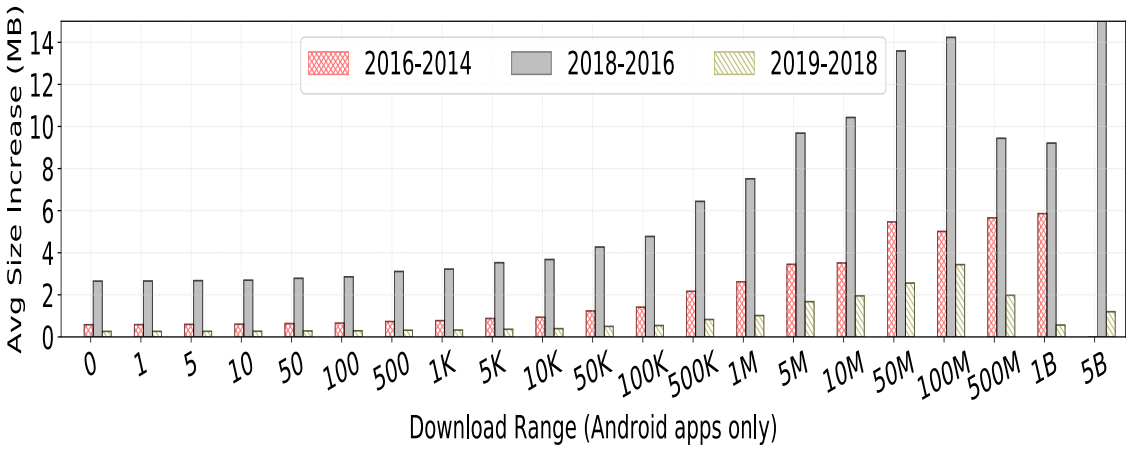


Figure 3.5: Longitudinal analysis of average increase in app sizes against their popularity (min download count). More increase is seen for popular apps, which suggests that as an app get popular, the developers pack more features in the same app to engage users.

to other apps. This behavior is captured by Figure 3.5. 931.9K common apps between 2014 and 2016 that we analyzed, 1,270 apps received 10-100M downloads and grew by 3 MB on average. Whereas, 18 apps received 1-5B downloads and grew by 6 MB on average in two years. Similarly, 356.2K common apps between 2016 and 2018, 16 apps received 500M-1B downloads and grew by 10 MB on average. We believe that once an app becomes popular (i.e., receive $>$ 500K downloads), more features are added to engage users, which result in more storage consumption. We further discuss this behavior in §3.3.

App size vs. device. Android supports many devices, each one is different (e.g., x86 vs. ARM, screen sizes). To create smaller and optimized apps, Google encourages developers to publish multiple APKs for the same app [21], each targeted to a particular device configuration. App Bundles [20] were also introduced in 2018 to allow developers to submit a single bundle (max 150 MB) of code/resources instead of submitting multiple APKs to target different devices. During installation, Google then selects appropriate APKs or generate on-the-fly from app bundles based on the user device configuration so that only necessary code/resources are downloaded. Apps that support optimized per-device APKs do not show a fixed installation size on their Google Play Store webpages; instead the size varies with each device [21]. To determine apps that support optimized APKs, we built a Soup [32] scraper to collect generic metadata from Google Play Store app webpages and checked installation sizes of apps. We found that only 58.3K (or 3.4%) of 1.7M apps that we analyzed from 2019, support per-device APKs. 4.3K of such device-specific apps received at least 1M downloads. Whereas, the remaining 96.6% of apps were *universal*; that is, developers create a single app to target multiple device types, resulting in apps that consume unnecessary storage. 22.6K of such universal apps were downloaded at least 1M times.

Summary.

- Both the number of apps and the average app size have been increasing over the years.
- The more popular an app becomes, developers pack more features to engage users, which result in more storage consumption.

• Large apps (>1 GB) are not limited to digital books or gaming, but spread across multiple categories, and are more popular than small apps.

• We believe due to technology advancements and increase in the maximum permissible app size, developers will continue to create feature-rich and graphically polished apps.

3.3 Static Analysis

This study performs longitudinal static code analysis of millions of free Google Play Store Android apps to report sources of storage consumption and understand how apps have evolved over time. We discuss our study methodology and findings here.

3.3.1 Research questions and methodology

From our static analysis, we sought answers to the following questions.

App composition. *What do modern apps consist of? In other words, what causes mobile apps to continue to grow in size? How has this behavior have changed over time?*

App resources. *What kind of assets and resources are used by apps? What effect do they have on app size?*

Third-party libraries. *What kind of third-party Java and native libraries are used by apps? How much do they contribute to app size?*

Methodology. We used state-of-the-art static analysis tools (e.g., apktool, dex2jar) to decompile the app APKs, and manually inspected the content and size of DEX files, native libraries, and any asset files present (e.g., icons). We further analyzed APK Java executables using LibScout [33] to determine third-party Java libraries being used. Similarly, we analyzed apps with OSSPolice [30] for native libraries being used.

Dataset. Our dataset consists of APK and OBB files of millions of free Google Play Store Android apps. It includes,

– 1.1M free apps collected in October, 2014 by PlayDrone [29].

– 1.6M free apps collected in December, 2016 by OSSPolice [30].

– 1.7M free apps that we collected in August, 2019.

3.3.2 Findings

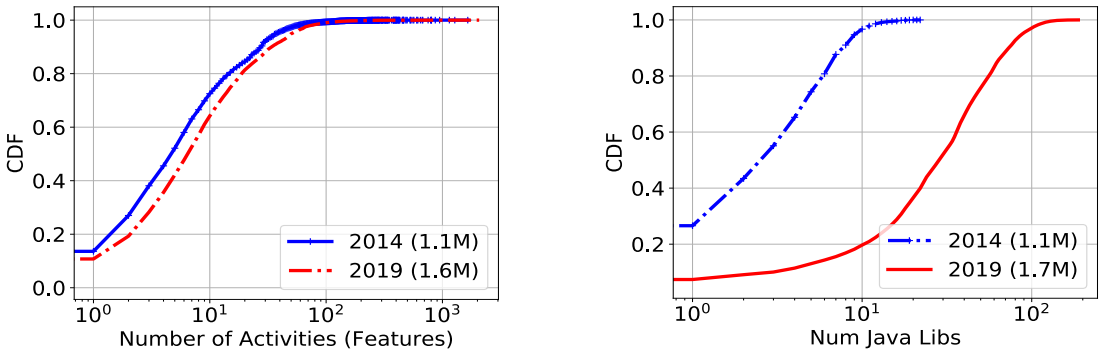
Here we report results from our static analysis of millions of Google Play Store Android apps in our dataset.

App composition. We found that modern apps are large and complex installation packages, consisting not only of executable binaries and libraries, but also various auxiliary resources, such as icons, images, animations, sounds, videos, text and xml files needed by the app to provide rich user experience. For instance, files under resource dir `res/mipmap/` of an app contains various icons needed by the app. Similarly, `res/values/strings.xml` file holds app strings. On Android, these auxiliary files are part of the app APK archive, and are not extracted on device during installation; instead, apps consume them directly from APK. We compared app APKs, and found a 27.23% and 32.71% increase in the number and average size of such auxiliary files, respectively from 2014 to 2019.

Universal Apps. To be able to target multiple disparate devices without duplicating the engineering effort, developers leverage increasing APK size limits to typically build and release universal apps. Such apps bundle auxiliary assets catering to more than one demographic region (e.g. text, fonts, icons, and sounds in various languages) and device type (e.g. ARM vs. x86, phone vs. tablet, small vs. large screen size). Of the apps we analyzed, a number of resource files target multiple device display types, such as xhdpi, mdpi, and h720dp. This corroborates our findings from §3.2.2 that suggests 96.6% apps from 2019 we analyzed had fixed one-size-fits installation APKs.

Similarly, we found that a host of apps contain auxiliary files that cater to more than one culture (e.g., Spanish) or language (e.g., Espanol or es_ES code). For example, files under `res/values-b+es/` dir contain icons for locales with the `es` language code and the ES country code Android resolves culture- and language-specific files at runtime depending

on the device locale settings for native user experience [34]. However, since device locale settings are rarely altered, a large number of such files are not used actively and end up consuming unnecessary storage space. We found a 46.4% increase in number of apps supporting resources specific to a demographic region in five years.



(a) *# App features (Activities).* (b) *# App Java libs.*

Figure 3.6: Static analysis of 1.1 and 1.7 million free Google Play Store Android APKs collected in 2014 and 2019, respectively.

App features. An Android app typically has multiple `Activities`. Each activity implementation is placed in a separate Java or C/C++ file and offers a particular feature or allows users to perform a specific feature (e.g., login, take photo). Since users interact only with Activities, we use them as a proxy for app features. Our analysis reveals that modern apps contain more features (or Activities). Figure 3.6 shows our findings. We found an average of 14.14 Android Activities per-app across 1.7M free Google Play Store apps collected in 2019, compared to only 10.18 Activities per-app in 2014.

Figure 3.7 shows increase in the number of Android Activities in top 30 Google Play Store apps selected at random, each with at least 5 million downloads. Compared to an average increase of 4 Activities per-app across all apps, we found an average increase of 100 Activities per-app across top 30 apps, which corroborates our findings from §3.2.2 on top apps. Specifically, as apps become popular developers pack more features and services within the same *super app* to provide one-stop experience to their users and

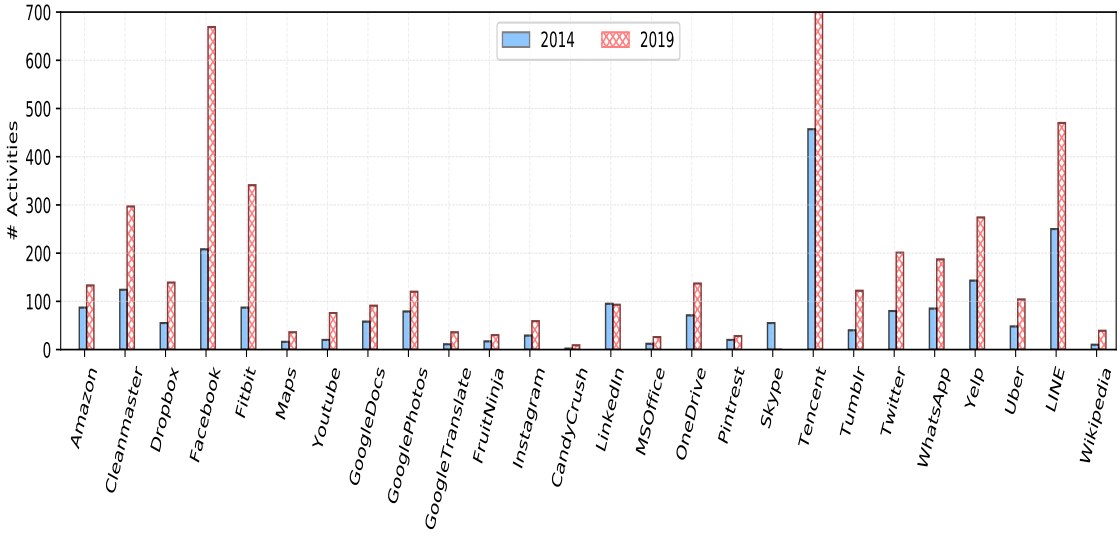
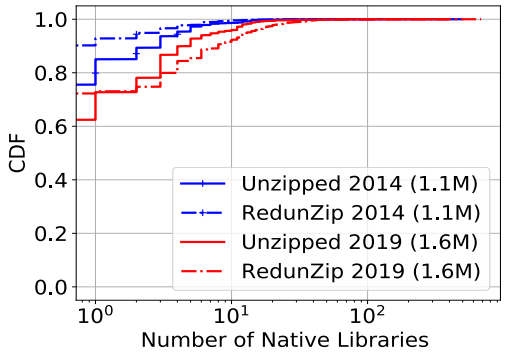


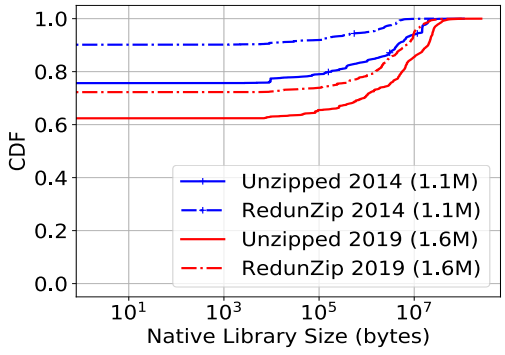
Figure 3.7: Increase in the number of Android Activities (features) in top (>10M downloads) Google Play Store apps collected in Oct'14 and Aug'19.

create an ecosystem. For instance, Tencent app added payments and ride sharing, among other services to their app. Consequently, such popular apps grow substantially more in size on average over the years compared to other apps. This behavior is further captured by Figure 3.5.

Java libraries. To quickly bring their apps to market, app developers often focus on the unique app features and workflows and rely on third-party libraries or Software Development Kits (SDKs) to import common features. We analyzed apps in our dataset with LibScout [33] to list JAVA libraries being used. We leveraged pre-compiled JAVA JAR files from Maven [35] and JCenter [36] to generate unique library profiles used by LibScout to compare app JAVA `classes.dex` files against, and produce a list of matching libraries found in apps. Our analysis show that modern mobile apps rely on third-party SDKs for a host of functionality, such as tracking analytics, debugging logs, displaying advertisements, and rendering PDFs. In particular, we found over a half of the free apps host in-app advertisements for monetization. Overall, we found that the average number of Java libraries per-app increased from 3.78 in 2014 to 34.76 in 2019, which adds to the storage pressure.



(a) Cumulative distribution of number of native libraries found across all apps.



(b) Cumulative distribution of sizes of native libraries found across all apps.

Figure 3.8: Analysis of native libraries found across 1.1 and 1.7 million free Google Play Store Android app APKs collected in 2014 and 2019, respectively.

SDK Code Bloat. Third-party SDKs implement generic functionality. For example, a cryptographic library may implement multiple encryption algorithms. However, only a subset of SDK *methods* is used by app developers, even though the entire third-party SDK is imported and distributed with the app, resulting in code bloat. Our LibScout analysis reveals that no method was used for 13.28% Java libraries in 1.7M apps from 2019. Furthermore, less than 40 methods were used for 40% of SDKs.

Redundant SDKs. Developers also add multiple third-party SDKs for the same functionality. For example, many apps integrate both Google and Facebook social media SDKs to offer easy account login functionality. However, the user may only use their favorite based on their preferences, rendering the code of other similar SDKs unused.

Native libraries. Besides importing JAVA libraries, app developers also use third-party native libraries. As mentioned in §2.1, native libraries are written in C/C++ and compiled to machine CPU. We found that both the number of apps containing native libraries and the average number of native libraries per app increased by 10% in five years. Figure 3.6b shows the distribution. Specifically, 290K (or 24.42%) of 1.1 million free Android apps from 2014, contain at least one native library in 2014. Average number of native libraries across those

Table 3.3: **Increase in native libraries supporting multiple CPU architectures.** The number of app with native libraries supporting multiple CPU architectures increased over time. However, sedundant cross-architecture `x86/x86 64`, `ARMv5/v8`, `MIPS/MIPS64` libraries will never be accessed on `ARMv7` devices.

CPU Arch	Number of apps (%)			Number of native libs (%)		
	2014	2016	2019	2014	2016	2019
ARMv5	63.08	59.13	44.10	32.40	24.16	8.86
ARMv7	74.75	78.76	94.35	53.16	42.71	38.41
ARMv8	0.04	8.26	42.39	0.02	3.54	12.95
MIPS	9.38	16.26	23.46	2.93	4.23	3.92
MIPS64	0.04	3.29	9.65	0.02	0.74	1.42
X86	24.39	46.73	67.94	11.16	21.31	24.85
X86_64	0.05	7.19	36.26	0.03	3.12	9.58
OTHER	0.35	0.23	0.08	0.28	0.18	0.02

290K apps was 3.2. In 2016, 640.7K (or 32.52%) of 1.97 million free apps contain at least one native lib, with an average of 6.3 native libraries per app. This number increased to 10 per app in 2019, with a total of 669K (or 37.7%) of 1.77 million apps containing native libraries.

We also found redundant cross-architecture `x86/x86_64, ARMv5/v8, MIPS/MIPS64` native libraries. Such libraries will never be accessed on `ARMv7` devices. We found 1.49 such libraries (consuming 10 MB) per app, on average. The number of redundant native libraries increased significantly in five years. We found 4 million such libraries across 669K apps in 2019, with an average of 6 per app. Whereas, the average number of native `ARMv7` libraries per app across all 669K apps was 4.

This suggests that despite Google's guidelines to create smaller and device-specific optimized apps [21, 20, 37], developers create universal apps. It is important to note that ARM devices are backward compatible; that is, an a native library compiled for `ARMv5` or `ARMv7` architecture is compatible on an ARMv8 Android device. Similarly, x86 Android devices can emulate ARM instructions using a binary translator [38] at runtime, trading performance and battery. However, developers include individual CPU-specific native

libraries to to provide improved performance and experience, trading more storage space.

Furthermore, 25% of the 290K apps from 2014 that contain native libraries were found to contain at least one *unstripped* native library. Such unstripped libraries contain ELF debug information and full symbol table. By default, the Android development environment (Android Studio) removes (strips) all debug information that may be embedded in native libraries by compiler. Developers may disable the stripping feature it for easy debugging during development at the cost of bigger libraries, and re-enable for a lean release. In 2019, 2/3 of apps containing native library included at least one unstripped library.

Additional to debug info, native libraries also contain symbols for the linker to resolve references at runtime. By default, the compiler adds symbols for all *exported* functions that may potentially be invoked at runtime. This, however, increases library size needlessly, particularly if the app only invoke a small fraction of such functions at runtime. We found that about 88% of native libraries have more than 50 exported functions that may add to the library size. App developers can modify Android compiler flags (e.g., include `-fvisibility=hidden` for GCC) to reduce the number of such unnecessary symbols, generating smaller libs.

In-app products. Another source of increase in app sizes is use of in-app products. Many free apps allow users to purchase additional features or related products from within their apps. For instance, Disney Pet Palace book allows users to purchase more pets. Similarly, games allow users to purchase advanced levels. However, such features will only be accessed once paid for and unlocked. We found that about 118K apps in 2019 support in-app purchases, consuming unnecessary storage space.

Duplicate code. As mentioned previously, apps import host of third-party JAVA and native libraries (or SDKs) to implement common functionality. However, use of common SDKs result in code duplication. For example, we found that 28% to 90% of the 1.7M apps that we analyzed from 2019 contain the same version of at least one library package from `com.android.support`. 6.6% contain the same version of OkHttp library.

Table 3.4: **Evolution of top Android apps over time.** Change in sizes, number of features (or Activities), and number of native libraries across top 10 (each over 1 billion installs) Android apps from 2014 to 2019.

App	# Activtis		Size (MB)		# Libs	
	'14	'19	'14	'19	'14	'19
AmznKdle	65	123	30.83	37.29	2	54
AgryBrds	12	25	47.26	99.00	3	8
Dropbox	55	130	28.24	50.62	4	12
LinkedIn	53	104	28.07	29.44	2	9
Facebook	208	563	25.20	51.58	44	140
Tencent	489	900	27.58	105.70	58	212
Twitter	98	193	14.29	20.77	20	48
Yelp	138	263	14.60	24.30	10	80
Fitbit	87	250	16.63	57.44	0	208
Starbcks	6	56	7.80	42.75	12	92

Additionally, apps from the same vendor contain common code. For example, we found that Google Maps, Google Hangouts, and Google PlusOne apps contain the same version of `libcrashreporterer.so` library. Facebook and Instagram apps, being from the same vendor, contain the same version of core native libraries, such as `libvideo.so` and `libfb.so`.

To find duplicate native libraries across all apps from 2019, we extracted libraries from app APKs using `apktool`, and calculated `md5sum` for each library. We only analyzed `ARMv7` libraries as most (94.35% of 669K) apps contain such native libraries (see Table 3.3). We found 2.1 million (or 91.84%) duplicate `ARMv7` libraries, occupying a total of 4.48 TB of redundant storage space. Therefore, a simple file-level deduplication scheme could save 2.13 MB per app, on average.

Opaque Binary Blobs. Google Play Store imposes a limit of 100 MB on Android APKs [31]. However, developers can include up to two auxiliary OBB [24] files for creating bigger and more complex apps, such as graphics-rich 3D games. Each OBB file can occupy up to 2 GB of storage space. As such, they add to the storage pressure significantly. We,

therefore, inspected OBB files from 150 large (> 100 MB) Google Play Store Android apps across all categories with at least one million downloads to understand how developers use OBB files.

We found that OBBs are zip (compressed) archives of multiple small binary (e.g. shaders, textures) and multimedia objects (e.g. images, sounds, videos etc.). For example, 1497 MB OBB archive from Asphalt 8 Airborne, a racing game, contains 11,221 distinct objects, ranging from 100 bytes to 13.47 MB. Developers utilize OBBs to support hardware advancements, such as bigger screen sizes and enhanced resolution (e.g. 4K, 8K). Games and digital books developers also leverage OBBs to hide proprietary implementation (e.g. mesh and textures), causing them to become particularly storage-heavy [39].

Developers also abuse OBB files to build universal apps, bundling auxiliary assets catering to more than one demographic region (e.g. text/audio in various languages) and device type (e.g. ARM vs. x86, phone vs. tablet). For example, OBB archive from Nightmares from the Deep game contains 445 icons and images containing non-English content (totaling 10 MB).

Summary.

- Modern apps pack more features and third-party SDKs/libraries. Number of features in top apps doubled in five years, resulting in bigger apps.
- Most apps are universal, containing cross-architecture native libraries for improved performance on every device type at the cost of more storage.
- Apps also contain paid features that are only unlocked once the user purchases them. Nevertheless, such features consume storage space regardless.
- Installation of apps that use common SDKs or are from the same vendor result in duplicate SDKs and libraries, consuming redundant storage space.

3.4 Install-time behavior.

This study focuses on installation-time storage consumption of Android apps.

3.4.1 Research questions and methodology

From our install-time study of apps, we sought answers to the following questions.

Install-time consumption. *How much storage do mobile apps consume upon fresh installation (no usage)?*

Sources of consumption. *What causes this additional storage consumption?*

Dataset. To evaluate app storage consumption during fresh installations on mobile devices, we analyzed 30 top Google Play Store Android apps from 2019, each with at least 10M downloads.

Methodology. We performed fresh installation of each of the apps in our dataset on a LG Nexus 5 device, and used Android Debug Bridge (ADB) shell scripts to inspect the size of each file in all app storage areas without using launching the app.

3.4.2 Findings

Storage consumption. Figure 3.9 shows our findings. We found that upon installation, 27 apps expand to further consume at least 1.5x storage space of their respective package. Facebook app, in particular, expands from 41.79MB package to consume sizable 214MB (over 5x increase) when installed.

OAT creation. Common cause of additional storage consumption during installation process is creation of additional files. During installation all JAVA `classes.dex` executable files are parsed by the ART/Dalvik JVM to create a single performance-optimized native executable content, called OAT file that can execute without the need for further JIT compilation, resulting in better user experience at runtime. Refer to §2.4 for details on the app installation process on Android.

OAT file is persisted on device under `/data/dalvik-cache/` app directory. Once an app is installed, compressed APK file is also stored on the device in the internal app-private directory at all times in order for the app to directly access its asset files as well

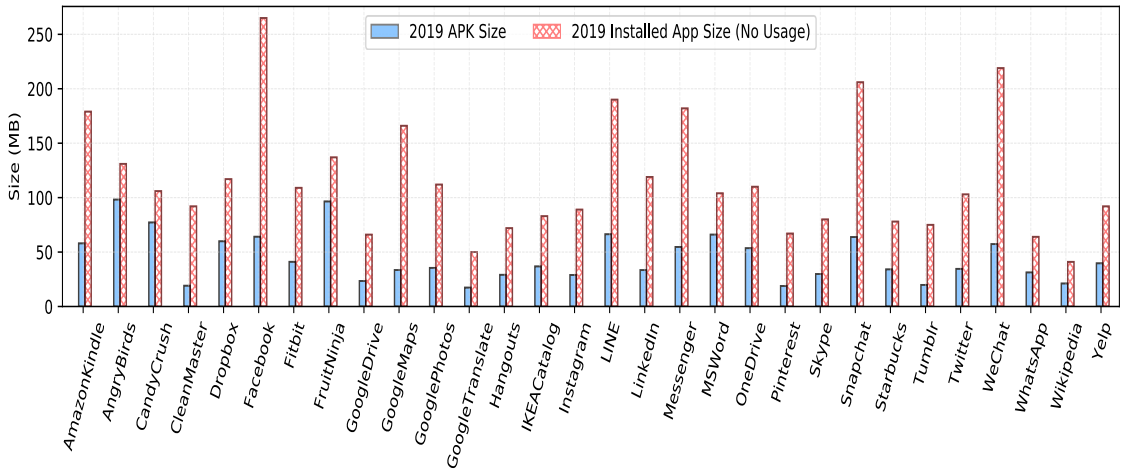


Figure 3.9: Install-time storage consumption analysis of top (>10M downloads) Google Play Store Android apps collected in Aug'19. Upon installation, apps expand to further consume additional storage.

as for the system to perform delta updates to the app. This results in additional storage consumption as two copies of each `classes.dex` file is created, namely one in the APK archive (compressed), and the pre-compiled OAT file.

Unzipped libraries. During installation, native libraries are also extracted on the device for faster I/O at the cost of more storage consumption. For example, Facebook APK extracts over 100 native libraries from `libxz` archive, consuming additional 50MB.

Additional persistent files. We found that apps also create `databases` and xml files under `shared_prefs` upon installation to host app settings. Apps that need auxiliary OBB files as a part of their functionality, download such files upon first activation (usage) of the app, not during installation. A few apps also download or create additional resources separately as needed. For instance, Facebook app preallocates storage space by creating a large 20MB file, which explains its sizable expansion upon installation.

3.5 User study

To gain insight into the runtime storage consumption behavior of today's mobile apps, we carried out an IRB-approved study of Android users in the wild.

3.5.1 Research questions and methodology

Specifically, the purpose of the study was to answer the following questions.

1. **Storage consumption:** *How many apps do users install on average? How much storage do these apps consume after weeks (or months) of usage? How much is consumed by user data (e.g., photos, videos, docs, etc.)? How frequently users run out of storage space on the device?*

2. **Storage usage:** *How many apps are used daily on average? How many apps features do users interact with daily on average? How many executables (e.g., libs) and auxiliary files (e.g., icons) stored during installation are actually used? What are some interesting storage usage patterns?*

Participants. Our study was conducted with participants recruited via PhoneLab [40], a mobile testbed at the University of Buffalo. PhoneLab allows researchers to deploy and test Android changes with 100–300 participants, who are university affiliates (i.e., faculty, staff, students) across diverse groups of age, gender, and occupation [41]. We provided our Android changes to PhoneLab administrators, who then made our study available to their participant pool to volunteer.

Our initial sample comprised of 231 PhoneLab participants. However, data from 91 participants were removed due to substantial missing data. The final sample used in this study comprised of 140 participants, who provided complete data across 70 days.

OS and device. PhoneLab provided a 16 GB LG Nexus5 device to each participant, which they use as their primary smartphone for the study duration, allowing us to collect representative usage data.

Privacy. To ensure the privacy of participants, PhoneLab anonymized all device identifiers when collecting data. Approval from the Institutional Review Board was obtained prior to the study.

Operating system. The device ran Android Lollipop version 5.0.1, customized to collect data from the Android `Logcat` memory buffer [42]. Data collected was stored on the device and securely uploaded to PhoneLab servers over HTTPS every night when the device was connected to WiFi.

New tracing tool. Smartphones are highly personal consumer devices. As such, each user persona type (e.g., gamer vs. non-gamer) is likely to have a completely different device usage patterns. Therefore, we built a novel context-aware storage tracing tool, called COSMOS, that not only collects low-level file system events, but also apps user interact with, along with various contextual attributes (e.g., location). It consists of a daemon process and a native library that is transparently loaded into each Android app using `LD_PRELOAD` when the app is launched. Upon initialization, the library hooks relevant file system APIs (e.g., open, read) in `bionic` C library by registering wrappers functions that intercept and log file system requests. Table 3.7 summarizes APIs we hook and data we collect. This design requires no change to individual apps or the Android framework. We deployed COSMOS on each participant's device as a background system service that posts data to the Android Logcat buffer, which is collected and uploaded by the PhoneLab framework.

We have designed COSMOS particularly for lightweight continuous data collection on mobile devices. For instance, we avoid periodic polling for device context in order to minimize the runtime overhead; instead, our daemon process subscribes to various Android system services to collect contextual attributes. Table 3.6 lists them. Similarly, our wrapper library allocates a large memory buffer in each app to allow bulk posting of file system events and minimize IO overhead. In contrast, existing file system monitoring tools such as inotify [43] pose high memory and performance overhead due to recursive watchpoints on each dir [44]. User-space file system tracing based on FUSE [28] (e.g., LoggedFS [45]) incur up to 4x performance overhead [46]. In-kernel tracing frameworks (e.g., ftrace) log all low-level file system requests, and thus require large kernel memory buffers. Additionally, requests from Android system services must be filtered to only capture storage accesses

Table 3.5: **Summary of contextual data collected from our user study.** We collect multiple spatio-temporal contextual attributes such as device location, and time of the day.

Attribute	Android Service	Context Retrieved
Timestamp	System Time	Time, Day, Month, Year
Location	LocationServices	Location Updates
UserActivity	ActivityRecognition	e.g., walking, etc.

Table 3.6: **Summary of device events we subscribe to for data collection from our user study.** We subscribe to various Android system services, and log events as they occur. This minimizes data collection overhead, compared to continuously polling for events.

Device Event	Android Service	Description
Bluetooth	BluetoothAdapter	Pairing Updates
Battery	BatteryManager	Battery Status
Cellular	ConnectivityManager	Connectivity Info
Wireless	WifiManager	Wifi SSID
PhoneStatus	AudioService, etc.	e.g., slient, etc.
Storage	StorageServer, etc.	Storage reclaimation

from apps.

Platform changes and data collection. We made changes to the Android core framework to continuously collect the following data from each participant.

1. Detailed low-level file system activity traces, and relevant parameters (e.g., path, size).

2. Detailed device and app usage statistics (e.g., app `Acitivity` invoked by the user). We made a small change to the core Android framework to track app Activity usage.

3. Stateful device updates (e.g., network connectivity, storage reclamation events).

4. Multiple spatio-temporal contextual attributes such as device location, time of the day, and physical activity of the user (e.g., on foot, in vehicle, etc.). Table 3.5 lists them.

We focused only on the storage consumption of the `/data` partition because it is fixed in size, shared among all apps, and hosts user data such pictures (see §2 for details).

3.5.2 Findings.

Here we present our findings on runtime storage consumption of modern mobile apps based on the data collected from the user study.

Table 3.7: **Summary of file system APIs hooked.** We hook multiple file system APIs such as `read()` and `write()` in `bionic` C library on Android to intercept and log all file system requests made by apps.

API	Input Parameters	Description
Read	File descriptor, Offset, Length	File being read from
Write	File descriptor, Offset, Length	File bring written to
Open	File descriptor, File path	File being accessed
Mmap	File descriptor, Access permissions	File being accessed

Storage consumption. We found that each user has an average of 122 apps installed on their device. Of those, 91 (or approx 75%) were pre-installed system apps. Figure 3.10 shows the breakdown of storage consumption of `/data` partition for 20 different users chosen at random. The space consumption of an app is calculated as the sum of sizes of its individual files found in various designated internal as well as external app storage locations (see Table 2.3). Space consumed by the OS is calculated based on the files present in the internal `/data` partition that do not belong to any app. Documents, images, audio, and video files are identified by the types of files found in the corresponding designated dirs under primary external partition (or `/sdcard`). For instance, on Android devices, pictures taken using the digital camera on the device are stored under `/data/media/DCIM`, whereas apps save picture messages in `/data/media/Pictures`. Similarly, audio files are stored in `/data/media/audio`.

As seen in the figure, apps consume almost 50% of the total storage across users, on average. For some users, the consumption exceeds 60%. Apart from user data such as pictures, music, videos, etc. that are commonly known to consume significant storage space due to high-quality multimedia content, we found that over time modern apps also can occupy a significant storage space on smart mobile devices. For instance, over 63% or 8.44GB of storage is consumed by apps for user-7 and user-13, even though none of the installed apps are $>$ 1GB apps.

To get more insights into what causes apps to consume storage, we further studied the

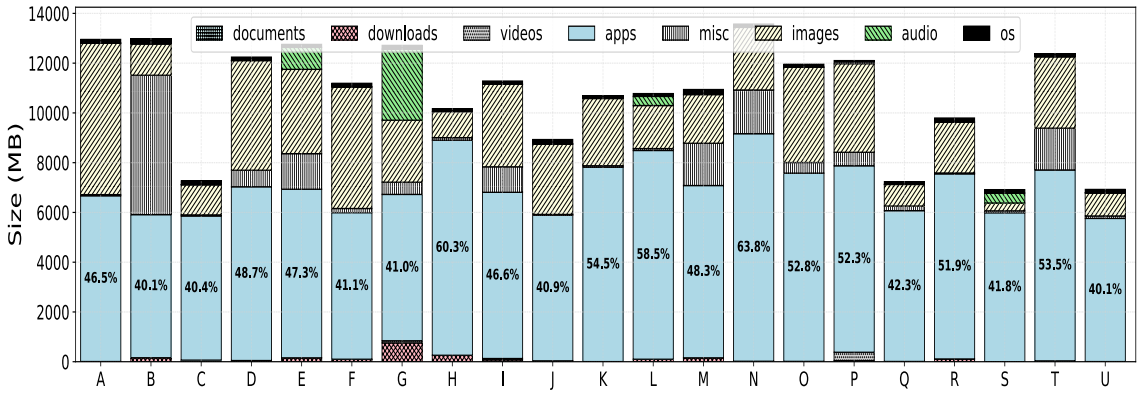


Figure 3.10: Breakdown of `/data` storage partition snapshot of randomly selected 21 users. The space consumption of an app is calculated from the sizes of its various individual files found in various internal as well as external app storage areas (see Table 2.3). Modern mobile apps consume a sizable chunk of storage over time.

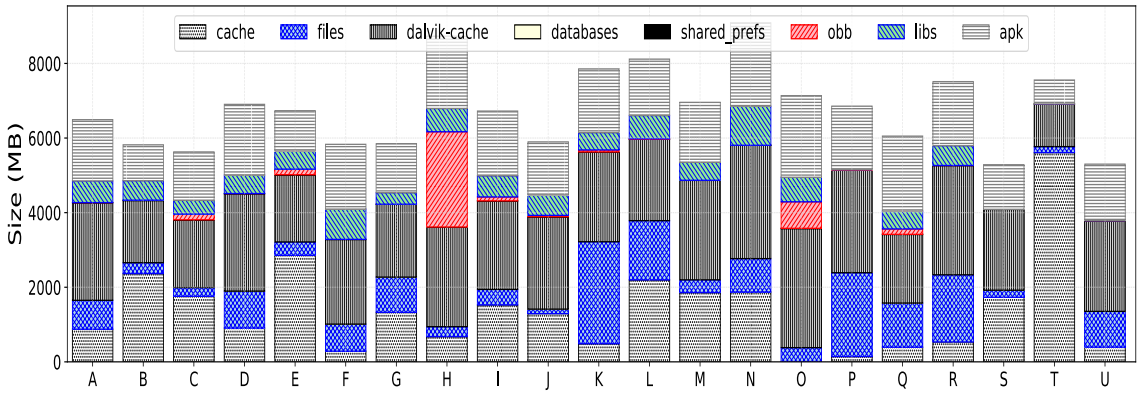


Figure 3.11: Further breakdown of storage consumption of apps in Figure 3.10 as per the app storage areas (see Table 2.3), namely files (F), cache (C), databases (D), shared preferences (S), native libraries (L), OBB files (O), APK files (A), and Dalvik cache (V) shows that apps frequently persist data on the device (e.g., caches, files, db, libs) to provide low latency and streamlined experience, particularly under fickle network conditions. Cache dir space is blindly reclaimed when the device runs low on storage regardless of device/apps usage pattern. In contrast, the space occupied by other persistent storage areas (e.g., files, libs) of an app is not reclaimed until the app is manually deleted.

storage consumption of apps as per various designated app storage locations on Android devices (see Table 2.3). Figure 3.11 shows the breakdown of storage consumption. We found that on average 20% and 11.86% of the storage is consumed by app caches (C) and files (F), respectively across 21 users. This suggests that app developers freely exploit persistent storage by creating persistent files as needed and frequently caching or pre-fetching content from the Cloud to improve app runtime performance and offer streamlined user experience, particularly over irregular networks. As a result, through the daily use of apps, mobile devices accumulate large amounts of data and quickly exhaust storage space.

Yet, apps are not constrained by storage consumption limits. A single app is allowed to consume up to 90% of storage on Android version 8.0 [10]. App caches (C) are temporary, and are reclaimed automatically when the overall device consumption reaches the 90% threshold on Android. Nevertheless, blind removal of cached working set not only affects the performance of frequently used apps, but also for such apps the caches are quickly repopulated on next usage. For example, for user-1 storage space occupied by Fitbit cache is cleared, but is repopulated the same day Figure 3.13.

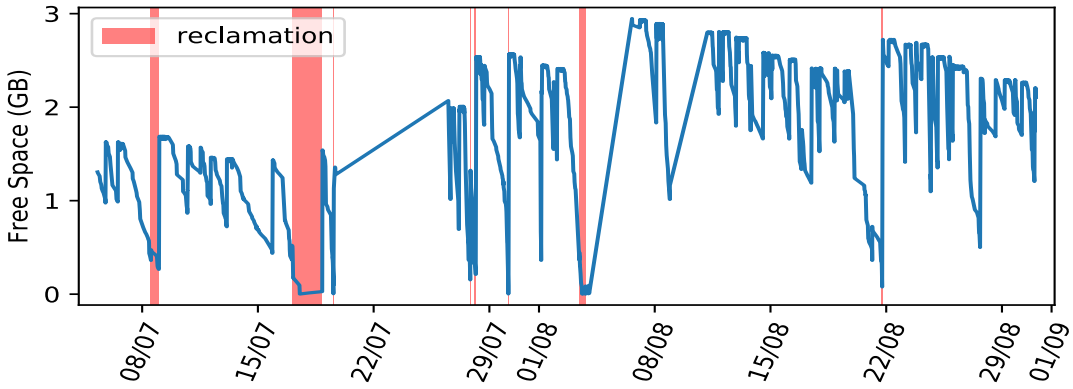


Figure 3.12: **Free space available on the device for user-O over time.** As the storage consumption on the device reaches 90%, the system starts to reclaims storage space by blindly deleting app caches. However, due to extensive device usage, app caches are repopulated quickly. The regions shaded in red represent various reclamation periods.

Unused apps and feature bloat. While our static analysis (§3.3) shows that modern apps pack 2x more features, findings from our user study shows that app usage follows the

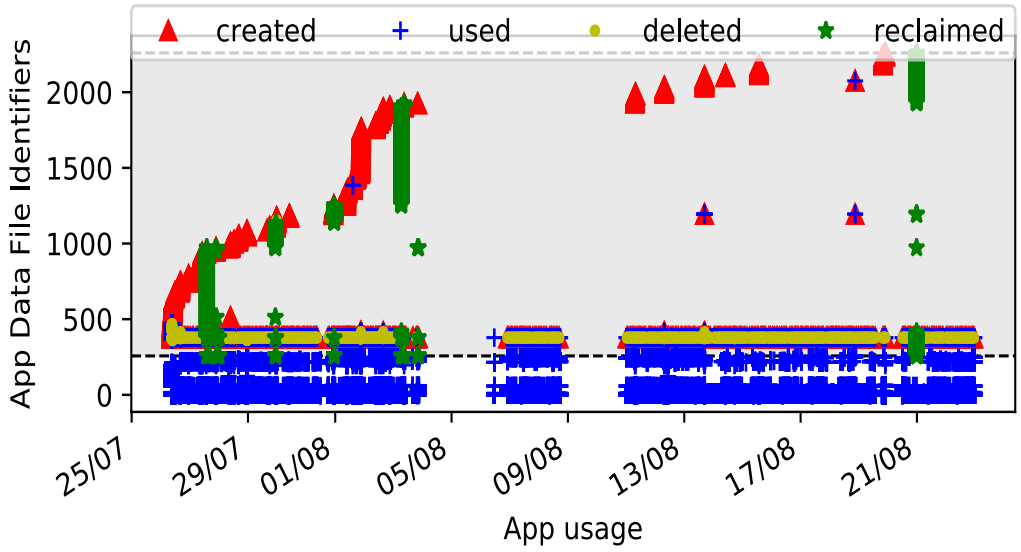


Figure 3.13: **Fitbit app storage consumption over time for user-O.** The area shaded in grey represents Fitbit app cache on the device. As the app is used, cached files are created (shown as red triangles) and get accumulated over time. However, only a small fraction of files created are actually used (shown in blue plus symbols) and only a small set of cached files are deleted by the app (shown in yellow dots). When the storage on the device falls below a threshold, storage space occupied by cached files is reclaimed (depicted by green stars). Due to blind reclamation of cached app data, files in the working set are recreated as and when needed.

Pareto Principle; i.e., only a small fraction of app features are actively used. We found this behavior to be consistent across most of the study participants. Table 3.8 summarizes our findings. For example, Facebook APK installs over 120 native libraries. However, we found that for most users less than 50 such libraries were used over the course of 70 days of its usage. LinkedIn app contains four native libraries. However, only three were used in 70 days of its usage for most users. Similarly, we found that a lot of app Activities installed as a part of the app are not used during its usage. As shown in Table 3.8, most users interact with less than 10% of overall the app Activities. For example, apps integrate multiple third-party social media SDKs (e.g., Facebook) to offer easy account creation functionality. However, only one is used; code of remaining SDKs is never executed. Furthermore, once the user creates an account, the related account creation functionality is not longer required. Many free apps (e.g., games) offer in-app purchases of additional features. Such features are only accessed once paid for and unlocked. This reveals that modern apps are heavily bloated:

90% number of app executables and third-party libraries that are persisted on device and optimized for performance during installation (see §3.4), are not actively used.

Furthermore, we found that app usage is highly correlated with user context. For example, user-5 uses *maps* app only once a week, every Sunday. Similarly, user4 uses *yelp* app only on weekends.

Table 3.8: Snapshot of storage consumption on devices across 21 users. System apps were pre-installed, unlike user-installed apps. Only a few apps are used daily. Furthermore, only a fraction of stored files are actively used.

User	Apps Installed			Apps Used		# Libs Used by Apps	
	Total	System	User	Avg Daily	Overall	Avg Daily	Overall
A	135	97	38	17	45	2	4
B	113	95	18	11	32	3	5
C	116	93	23	15	34	2	7
D	123	95	28	8	23	4	6
E	121	97	24	10	37	2	4
F	130	97	33	16	36	2	5
G	107	96	11	15	31	4	6
H	115	91	24	18	37	2	6
I	166	102	64	13	51	3	8
J	124	92	32	17	39	3	7
K	122	90	32	12	34	1	5
L	129	96	33	12	34	3	9
M	132	98	34	19	59	5	8
N	145	98	47	11	35	8	11
O	146	99	47	15	41	2	5
P	122	96	26	12	35	1	4
Q	131	91	40	18	30	2	6
R	145	99	46	8	41	3	8
S	121	89	32	14	50	4	4
T	117	92	27	10	25	5	7
U	123	94	29	16	45	3	7

Data hoarding. Apart from freely caching temporary data, our analysis suggest that apps also hoard persistent data as needed, such as auxiliary files and databases, analytics to track user engagement, crash reports for debugging, and advertisements for monetization.

As such, for efficient device storage management, it is important to compare these numbers with corresponding install-time stats (i.e., no usage) and understand how much storage is typically consumed by data hoarded by apps over time. To do so, we performed fresh installation of the same apps and manually analyzed the sizes of files created each app under `/data/data/app` storage directories without launching the apps (see Table 2.3. To only analyze app data, we ignored all app executables (OAT, native libraries), files under `cache`, and other files extracted from app APK. We found that app data could consume up to 4.6GB, and is never deleted automatically. Worse yet, unlike cached content (i.e, files in `cache` dir) that is reclaimed automatically when the device runs low on storage, the space occupied by persistent files residing in other storage areas (e.g., `/files`) is not automatically reclaimed when the device runs low on storage. Such persistent data requires manual deletion on Android and app uninstallation on iOS.

Files under public-readable `sdcard` are not deleted even upon app uninstallation. During our analysis of device storage of users in our study, we found that games in particular, download high-definition in-app video advertisements stored on device for weeks. We were surprised to see over 50 MB of month-old video (mp4) and image (png) advertisements from AdColony, UnityAds, and Chartbooster. We detected several months-old advertisements and residual files, consuming over 680MB on average.

Redundant metadata. We found that functionally similar apps operating on a same data create their own copy of metadata, resulting in unnecessary storage consumption. For example, various photo viewing and editing apps, file browsers, and Cloud-based personal storage apps, such as Dropbox, Box, and Google Photos allow users to browse stored photos and videos. For better performance, each app generates their own thumbnails of all videos and photos stored on the device and the Cloud. This results in redundant metadata, which can occupy significant storage space depending on number of metadata objects created.

3.6 Mobile storage management

Here we summarize the findings from our study of modern mobile apps, and present a case for an automated storage management on mobile devices.

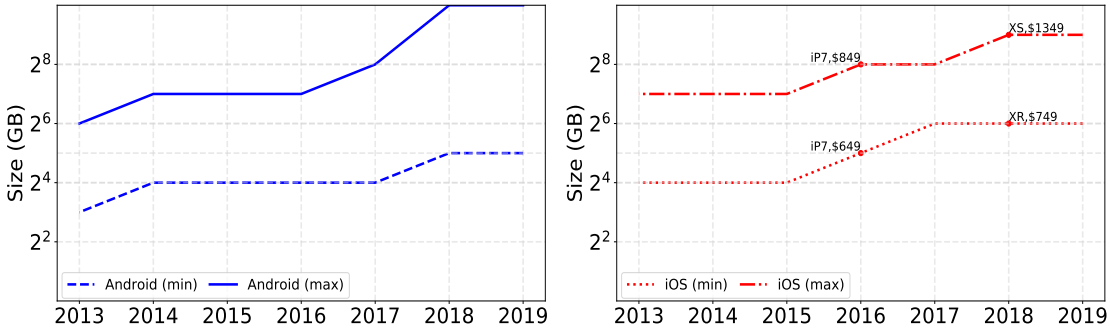


Figure 3.14: Growth in mobile storage capacity.

Storage demand vs. capacity. Our longitudinal study shows that modern mobile apps have evolved as large feature-rich monolithic packages: an app from 2019 packs 2x more features and consumes 2x more storage (up to 4 GB) than that from 2014 (see Figure 3.7). Furthermore, today's storage-heavy apps are not limited to games or e-books, but span across multiple categories. We believe that this trend will continue, particularly as the mobile technology (e.g., 5G) evolves and richer, more immersive apps supporting Augmented Reality, Artificial Intelligence, and 4K graphics are introduced.

To accommodate increasing demands, smartphones have seen growth in storage capacity. For instance, the minimum storage on iPhone quadrupled in five years: from 16 GB [47] in 2014 to 64 GB [19] in 2019. Figure 3.14 shows the trend. However, low-end budget devices are still prevalent. In fact, according to Counterpoint Research [48], the global average storage capacity of smartphones sold in 2019 was about 54 GB and 134 GB for Android and iOS smartphones, respectively. Since Android holds about 86% of global smartphone market [49], we deduce that conservatively at least 43% of the global smartphone users owned devices with less than 64 GB of storage in 2019. Such devices are more common in developing countries [50], and severely limit user experience [12, 11, 6].

Storage capacity vs. bloatware. While users can choose to pay a premium price for additional mobile storage [19], our analysis shows that today's mobile OSes and app development model is not optimized for storage. A high percentage of storage is consumed by OS and pre-installed apps. (shown in Figure 3.10). Modern apps are monolithic and universal: they 1) pack more Java code, which is precompiled to native machine code for best runtime performance trading 1.5x storage (see Figure 3.9), 2) contain cross-architecture libs for native performance on every device type (see Table 3.3). Furthermore, developers freely abuse persistent storage to cache data for streamlined user experience, host ads and user analytics for monetization. As such, today's apps are heavily optimized for performance. The assumption is that all apps/features are equally important to all users at all times. However, our user study shows otherwise: only 10% of apps/features are actively used §3.5.

3.6.1 Design space: challenges and opportunities

In the light of aforementioned findings, there is an increasing need for an efficient storage management system on smart mobile devices. Here, we discuss a few design opportunities and challenges.

1. Cloud augmentation. There exists a number of personal Cloud storage services, such as Dropbox[13], Box [14], iCloud [16], and Google Drive [15] that offer virtually unlimited storage to host user data for easy and ubiquitous access to data across multiple devices, under a tiered pay-as-use pricing model. As such, the Cloud offers a naturally attractive solution to the storage constraints on mobile devices. In fact, iOS v11 provides an optional feature to offload infrequently used apps to iCloud to help users make the most of the available iPhone storage [17]. However, it only removes the core iOS app, while retaining all its settings and data (e.g., login credentials) on the device for future stateful accesses. Nevertheless, offloading all app data will also offload unwanted data, such as crash logs, user analytics, video advertisements that apps hoard on the device, as shown by our user study findings §3.5. As such, blindly backing everything to the Cloud merely shifts the problem to managing

storage space in the Cloud, and users still need to perform manual management to delete data or pay for usage.

2. Storage quota. One way to address storage management challenges would be to introduce app storage quota and limit storage consumption per app. While this design will address the problem of apps freely abusing local storage to cache and hoard data (e.g., analytics, ads), identifying appropriate storage quota limits is challenging. Static limits will not scale as more functionality is introduced and apps become larger in size. Additionally, it requires developers to redesign apps to work with limited quota with no discernible impact on user experience.

Elastic quota model [22] overcomes these shortcomings by hard-limiting only persistent data and allowing temporary data to grow or be reclaimed depending upon the available storage space. However, it requires user to identify when and what data should persist or thrown away. This is because our user study findings suggest that not all apps are equally important to user at all times. In fact the relative importance of apps not only varies with the user (e.g., gamer vs. non-gamer), but also varies for the same user depending on the context (e.g., home vs. office).

3. Automated management. Our study shows that users typically engage with only a small fraction (10%) of installed apps daily, on average, and follow a fixed, repetitive schedule (e.g., yelp usage only on weekends). Therefore, we assert that modern OSes must intelligently and proactively manage the limited storage resources on behalf of the user depending on user's context-sensitive storage requirements. Multiple techniques, such as compression, deletion, content adaptation, and Cloud-backed hierarchical management could be leveraged to reclaim storage space consumed by contextually unwanted apps/data.

App code. As reported in section 3.4, upon installation, apps further expand to consume at least 1.5x storage space of their respective package due to generation of pre-compiled OAT files for better runtime performance. However, such performance-optimized OAT files of contextually unwanted apps could be deleted temporarily, falling back to regular

(non-optimized) executable code if needed.

Similarly, our findings suggest that not all app files and features are used at all times. For instance, many apps offer in-app purchases of additional features. Such features will only be accessed once the user has paid for and unlocked them. Depending upon the usage, only a few Java classes and their corresponding resource files (e.g., images, sound files, etc.) will be accessed. Consequently, data could be adapted to optimize for storage. Depending on the content type (e.g., text, exec, multimedia, etc.) appropriate adaptation scheme may be chosen. For example, high resolution multimedia files could be adapted into low resolution objects if deemed contextually unwanted. Only actively used Java executable classes could be optimized, reducing the size of the generated OAT files, potentially offering similar performance. Content in universal apps, such as cross-architecture `x86` native libraries on `ARMv7` devices and resource files in multiple languages can be safely deleted once deemed unnecessary.

Furthermore, common files across apps can be consolidated using deduplication. For example, we found 2 of the top 30 apps use same version of Joda-Time SDK to implement timezone functionality. Microsoft Skydrive app and Fruit Ninja game contain same version of `crashlytics` SDK native library, each consuming 121 KB and 185 KB of storage space on ARM and x86 architectures, respectively. Similarly, Dropbox and Snapchat apps contain the same version of `librsjni` library, each consuming 50KB. Overall, we found that a total of 30 MB of storage could be saved by simple file-level deduplication of native libraries found in the APKs of top 30 apps.

App data. Our findings show that apps freely use persistent storage to offer streamlined user experience by frequently caching and hoarding data. Compression can be employed to transparently reclaim some storage space occupied by app persistent data. Additional savings could be achieved by deleting cached or temporary data.

User data. A large portion of storage consumed by infrequently accessed data could be freed by offloading it to the Cloud. For example, user data such as pictures, audio, videos,

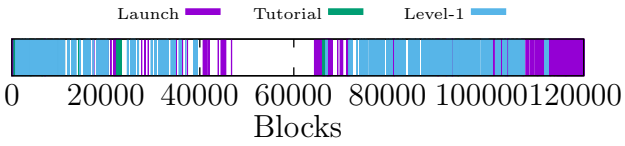


Figure 3.15: OBB accesses across three different levels

and documents could be backed up to existing personal Cloud storage (e.g., Dropbox, iCloud). Such data can be accessed on demand.

OBB files. OBB files in Android apps, being large archives, provide many opportunities for context-sensitive management. We monitored accesses to OBB files during app usage by leveraging file system tracing hooks. We found that different objects inside monolithic OBB archives are accessed at different times, depending upon the usage. For instance, games contain multiple levels. Every level accesses only a few objects (associated with it) to display on the screen. Figure 3.15 shows accesses to a 475 MB OBB archive from three different levels of Nightmares from the Deep, an adventure game containing 1,177 objects. As seen from the figure, a markedly different chunk of the OBB archive is accessed by different levels of the game.

Each game level demands different playing skills for users ranging from beginners to experts. As such, not all levels are accessed simultaneously. Depending upon the user's expertise, different levels and features associated with them will be accessed at different times. For inexperienced users, level with the highest difficulty may even never be accessed. Similarly, experts may no longer access initial levels if the player is far through the game.

Similarly, due to the complexity and graphical richness, these large apps contain interactive tutorials for improved user learning and engagement. Some apps, such as adventure games and digital books targeting kids, also contain story narratives to build context and provide background information. However, these tutorials and narratives may never be accessed once the user gets acquainted with the features. Therefore, depending upon particular usage of an app, a significant chunk of storage occupied by OBB files could be freed and downloaded again from the app store when needed.

CHAPTER 4

EXTENSIBLE USER FILE SYSTEM FOR STORAGE MANAGEMENT

Our study findings suggest that multiple storage reclamation techniques, such as deduplication, compression, and hierarchical management could be leveraged to yield significant storage savings on smart mobile devices. To be able to support such techniques, we introduce an extensible user file system framework that offers the native performance of kernel file systems, while retaining the safety properties of user file systems. Here we describe its design and architecture, and evaluate its performance with synthetic benchmarks as well as real-workloads on Android.

4.1 Overview

File systems implements low-level functionality and provide generic high-level abstractions to applications for accessing data. There are two schools of thought on developing new file system functionality.

Kernel file system. The first school advocates for kernel integration to achieve native performance. A number of different types of kernel file systems have been made available over the years. Examples include UNIX file systems [51, 52, 53], network file systems [54], distributed file systems [55, 56], and stackable file systems that add specialized functionality such as encryption [57], and tracing [58] to the host file system. Nevertheless, kernel implementation not only requires domain expertise, but also goes through time-consuming iterations of development and quality assurance. Worse yet, security vulnerabilities and bugs [59] in the implementation can crash the kernel and render the system useless.

User file system. The second school maintains the idea of minimizing the complexity in the kernel and proposes to host file system services as user processes. hosting file system services as user processes to minimize the complexity in the kernel. Micro-kernels [60,

61, 62] adopted the latter design. Living in the user space, such services not only offer better security (i.e., unprivileged execution) and reliability [63] when compared to in-kernel implementations, but also ease the development, maintenance, and debugging of file systems. Furthermore, third-party libraries can be reused for quick experimentation and prototyping. Therefore, many approaches to develop user space file systems have also been proposed for monolithic Operating Systems (OS), such as Linux and FreeBSD. While some approaches are specialized, targeting specific systems [64, 65, 66], a number of general-purpose frameworks for implementing user file systems also exist [67, 68, 69, 70, 28]. FUSE [28], in particular, is the state-of-the-art framework for developing user file systems. Over a hundred FUSE file system have been created in academic/research [71, 72, 73, 74, 75, 76], as well as in production settings [77, 78, 79, 80].

However, supporting multiple storage reclamation and data reconstruction operations with varying degrees of complexity (e.g., compression, hierarchical management) in the kernel will result into a system with questionable reliability. On the other hand, as discussed above, a user space implementation will not only offer better reliability, but also allow reuse of existing third-party user-space libraries (e.g., zlib), lending to the ease of development. Therefore, we designed ANODYNE file system as a stackable user file system. That is, it implements all file system functionality in user space and uses the underlying (host) file system to manage its data.

FUSE. However, being general-purpose, the primary goal of the existing user file system frameworks such as FUSE is to enable easy, yet fully-functional implementation of file systems in user space supporting multiple different functionalities. To do so, FUSE implement a minimal kernel driver that interfaces with the Virtual File System (VFS) operations and simply forwards all low-level requests to user space. For example, when an application (app) makes an `open()` system call, the VFS issues a `lookup` request for each path component. Similarly, `getxattr` requests are issued to read security labels while serving `write()` system calls. Such low-level requests are simply forwarded to user space.

While such a general-purpose design offers flexibility to developers to easily implement their functionality and apply custom optimizations, it also incurs a high overhead due to frequent user-kernel switching and data copying. For example, despite several recent optimizations made to FUSE to reduce user-kernel switching and data copying, even a simple passthrough FUSE file system can introduce up to 83% overhead on an SSD [81]. As a result, some FUSE file systems have been replaced by alternative implementations in production [77, 82, 83]. For instance, Android v7.0 replaced the sdcard FUSE daemon with an in-kernel implementation [77] after several years.

ExtFUSE. To overcome the performance limitations of existing user file system frameworks, we developed a novel extensible user file system framework, called ExtFUSE, for developing user file systems for UNIX-like monolithic OSes. It is based on FUSE, and offers the performance of kernel file systems, while retaining the safety properties of user file systems. It allows the unprivileged FUSE daemon processes to register "thin" extensions in the kernel for specialized handling of low-level file system requests, while retaining their existing complex logic in user space to achieve the desired level of performance.

The registered extensions are safely executed under a sandboxed eBPF runtime environment in the kernel (§4.2), immediately as requests are issued from the upper file system (e.g., VFS). Sandboxing enables the FUSE daemon to safely extend the functionality of the driver at runtime and offers a fine-grained ability to either serve each request entirely in the kernel or fall back to user space, thereby offering safety of user space and performance of kernel file systems.

4.2 eBPF

ExtFUSE leverages extended BPF (eBPF) [84], an in-kernel Virtual Machine (VM) runtime framework to load and safely execute user file system extensions.

Richer functionality. eBPF is an extension of classic Berkeley Packet Filters (BPF), an in-kernel interpreter for a pseudo machine architecture designed to only accept simple

network filtering rules from user space. It enhances BPF to include more versatility, such as 64-bit support, a richer instruction set (e.g., call, cond jump), more registers, and native performance through JIT compilation.

High-level language support. The eBPF bytecode backend is also supported by Clang/LLVM compiler toolchain, which allows functionality logic to be written in a familiar high-level language, such as C and Go.

Safety. The eBPF framework provides a safe execution environment in the kernel. It prohibits execution of arbitrary code and access to arbitrary kernel memory regions; instead, the framework restricts access to a set of kernel helper APIs depending on the target kernel subsystem (e.g., network) and required functionality (e.g., packet handling). The framework includes a static analyzer (called verifier) that checks the correctness of the bytecode by performing an exhaustive depth-first search through its control flow graph to detect problems, such as infinite loops, out-of-bound, and illegal memory errors. The framework can also be configured to allow or deny eBPF bytecode execution request from unprivileged processes.

Key-Value Maps. eBPF allows user space to create *map* data structures to store arbitrary key-value blobs using system calls and access them using file descriptors. Maps are also accessible to eBPF bytecode in the kernel, thus providing a communication channel between user space and the bytecode to define custom key-value types and share execution state or data. Concurrent accesses to maps are protected under read-copy update (RCU) synchronization mechanism. However, maps consume unswappable kernel memory. Furthermore, they are either accessible to everyone (e.g., by passing file descriptors) or only to `CAP_SYS_ADMIN` processes.

eBPF is a part of the Linux kernel and is already used heavily by networking, tracing, and profiling subsystems. Given its rich functionality and safety properties, we adopt eBPF for providing support for extensible user file systems. Specifically, we define a white-list of kernel APIs (including their parameters and return types), and abstractions that user file system extensions can safely use to realize their specialized functionality. The eBPF

verifier utilizes the whitelist to validate the correctness of the extensions. We also build on eBPF abstractions (e.g., maps) and apply further access restrictions to enable safe in-kernel execution, as needed.

4.3 Architecture

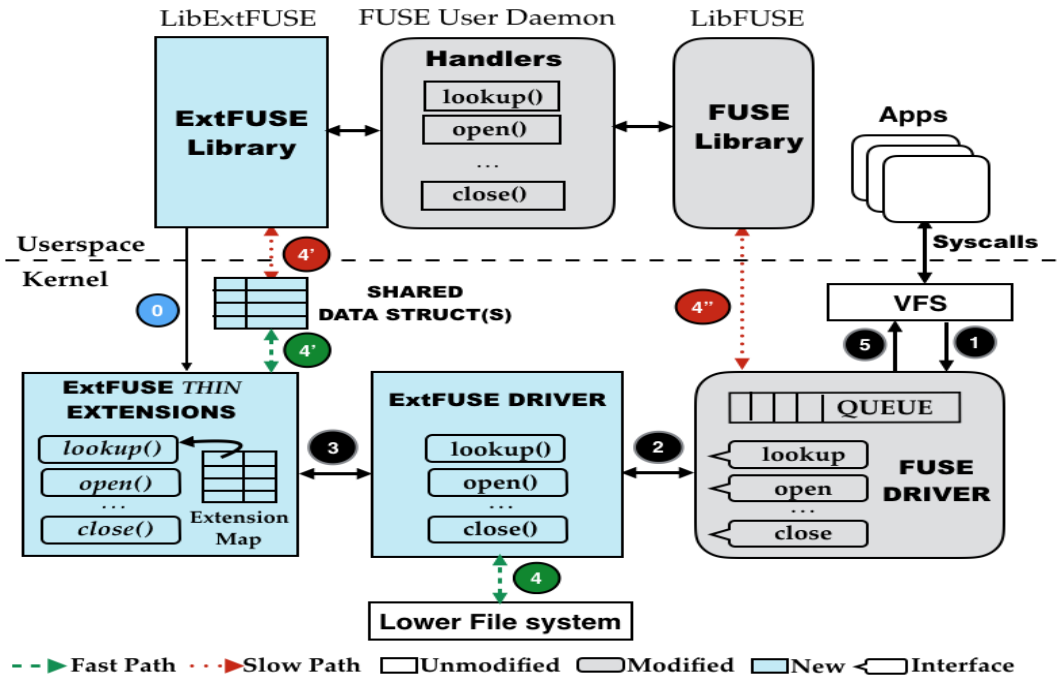


Figure 4.1: Architectural view of the ExtFUSE framework. The components modified or introduced have been highlighted.

Figure 4.1 shows the architecture of the ExtFUSE framework. It is enabled by three core components, namely a kernel file system (`driver`), a user library (`libExtFUSE`), and an in-kernel eBPF virtual machine runtime (`VM`).

The ExtFUSE driver uses interposition technique to interface with FUSE at low-level file system operations. However, unlike the FUSE driver that simply forwards file system requests to user space, the ExtFUSE driver is capable of directly delivering requests to in-kernel handlers (extensions). It can also forward a few restricted set of requests (e.g., `read`, `write`) to the host (lower) file system, if present. The latter is needed for stackable user file systems that add thin functionality on top of the host file system. LibExtFUSE exports

a set of APIs and abstractions for serving requests in the kernel, hiding the underlying implementation details.

Use of LibExtFUSE is optional and independent of `libfuse`. The existing file system handlers registered with `libfuse` continue to reside in user space. Therefore, their invocation incurs context switches, and thus, we refer to their execution as the *slow* path. With EXTFUSE, user space can also register kernel extensions that are invoked immediately as file system requests are received from the VFS in order to allow serving them in the kernel. We refer to the in-kernel execution as the *fast* path. Depending upon the return values from the fast path, the requests can be marked as served or be sent to the user-space daemon via the slow path to avail any complex processing as needed. Fast path can also return a special value that instructs the EXTFUSE driver to interpose and forward the request to the lower file system. However, this feature is only available to stackable user file systems and is verified when the extensions are loaded in the kernel.

The fast path interfaces exported by LibExtFUSE are the same as those exported by `libfuse` to the slow path. This is important for easy transfer of design and portability. We leverage eBPF support in the LLVM/Clang compiler toolchain to provide developers with a familiar set of APIs and allow them to implement their custom functionality logic in a subset of the C language.

The extensions are loaded and executed inside the kernel under the eBPF VM sandbox, thereby providing user space a fine-grained ability to safely extend the functionality of FUSE kernel driver at runtime for specialized handling of each file system request.

4.4 Workflow

To understand how EXTFUSE facilitates implementation of extensible user file systems, we describe its workflow in detail. Upon mounting the user file system, FUSE driver sends `FUSE_INIT` request to the user-space daemon. At this point, the user daemon checks if the OS kernel supports EXTFUSE framework by looking for `FUSE_CAP_ExtFUSE` flag in the

request parameters. If supported, the daemon must invoke LibExtFUSE `init` API to load the eBPF program that contains specialized handlers (extensions) into the kernel and register them with the EXTFUSE driver. This is achieved using `bpf_load_prog` system call, which invokes eBPF verifier to check the integrity of the extensions. If failed, the program is discarded and the user-space daemon is notified of the errors. The daemon can then either exit or continue with default FUSE functionality. If the verification step succeeds and the JIT engine is enabled, the extensions are processed by the JIT compiler to generate machine assembly code ready for execution, as needed.

Extensions are installed in a `bpf_prog_type` map (called *extension map)*, which serves effectively as a jump table. To invoke an extension, the FUSE driver simply executes a `bpf_tail_call` (far jump) with the FUSE operation code (e.g., `FUSE_OPEN`) as an index into the extension map. Once the eBPF program is loaded, the daemon must inform EXTFUSE driver about the kernel extensions by replying to `FUSE_INIT` containing identifiers to the extension map.

Once notified, EXTFUSE can safely load and execute the extensions at runtime under the eBPF VM environment. Every request is first delivered to the fast path, which may decide to 1) serve it (e.g., using data shared between the fast and slow paths), 2) pass the request through to the lower file system (e.g., after modifying parameters or performing access checks), or 3) take the slow path and deliver the request to user space for complex processing logic (e.g., data encryption), as needed. Since the execution path is chosen per-request independently and the fast path is always invoked first, the kernel extensions and user daemon can work in concert and synchronize access to requests and shared data structures. It is important to note that the EXTFUSE driver only acts as a thin interposition layer between the FUSE driver and kernel extensions, and in some cases, between the FUSE driver and the lower file system. As such, it does not perform any I/O operation or attempts to serve requests on its own.

Table 4.1: APIs and abstractions provided by EXTFUSE. It provides FUSE-like file system interface for easy portability. CRUD (create, read, update, and delete) APIs are offered for map data structures to operate on Key/Value pairs. Kernel accesses are restricted to standard eBPF kernel helper functions. We introduced APIs to access the same FUSE request parameters as available to user space.

FS Interface	**API(s)**	**Abstractions**	**Description**
Low-level	`fuse_lowlevel_ops`	`Inode`	FS Ops
Kernel Access	**API(s)**	**Abstractions**	**Description**
eBPF Funcs	`bpf_*`	`UID, PID, etc.`	Helper Funcs
FUSE	`extfuse_reply_*`	`fuse_reply_*`	Req Output
Kernel	`bpf_set_pasthru`	`FileDesc`	Enable Pthru
Kernel	`bpf_clear_pasthru`	`FileDesc`	Disable Pthru

DataStructs	**API(s)**	**Abstractions**	**Description**
SHashMap	`CRUD`	`Key/Val`	Hosts arbitrary data blobs
InodeMap	`CRUD`	`FileDesc`	Hosts upper-lower inode pairs

4.5 APIs and Abstractions

LibExtFUSE provides a set of high-level APIs and abstractions to the developers for easy implementation of their specialized extensions, hiding the complex implementation details. Table 4.1 summarizes the APIs. For handling file system operations, LibExtFUSE exports the familiar set of FUSE interfaces and corresponding abstractions (e.g., inode) for design compatibility. Both low-level as well as high-level file system interfaces are available, offering flexibility and development ease. Furthermore, as with `libfuse`, the daemon can register extensions for a few or all of the file system APIs, offering them flexibility to implement their functionality with no additional development burden. The extensions receive the same request parameters (`struct fuse_[in,out]`) as the user-space daemon. This design choice not only conforms to the principle of least privilege, but also offers the user-space daemon and the extensions the same interface for easy portability.

For hosting/sharing data between the user daemon and kernel extensions, LibExtFUSE

provides a secure variant of eBPF `HashMap` key/value data structure called `SHashMap` that stores arbitrary key/value blobs. Unlike regular eBPF maps that are either accessible to all user processes or only to `CAP_SYS_ADMIN` processes, `SHashMap` is only accessible by the unprivileged daemon that creates it. LibExtFUSE further abstracts low-level details of `SHashMap` and provides high-level `CRUD` APIs to create, read, update, and delete entries (key/value pairs).

EXTFUSE also provides a special `InodeMap` to enable passthrough I/O feature for stackable EXTFUSE file systems (§4.7.2). Unlike `SHashMap` that stores arbitrary entries, `InodeMap` takes open file handle as key and stores a pointer to the corresponding lower (host) inode as value. Furthermore, to prevent leakage of inode object to user space, the `InodeMap` values can only be read by the EXTFUSE driver.

4.6 Implementation

To implement EXTFUSE, we provided eBPF support for FUSE. Specifically, we added additional kernel helper functions and designed two new map types to support secure communication between the user-space daemon and kernel extensions, as well as support for passthrough access in read/write. We modified FUSE driver to first invoke registered eBPF handlers (extensions). Passthrough implementation is adopted from WrapFS [85], a wrapper stackable in-kernel file system. Specifically, we modified FUSE driver to pass I/O requests directly to the lower file system.

Since with EXTFUSE developers can install extensions to bypass the user-space daemon and pass I/O requests directly to the lower file system, a malicious process could stack a number of EXTFUSE file systems on top of each other and cause the kernel stack to overflow. To guard against such attacks, we limit the number of EXTFUSE layers that could be stacked on a mount point. We rely on `s_stack_depth` field in the super-block to track the number of stacked layers and check it against `FILESYSTEM_MAX_STACK_DEPTH`, which we limit to two. Table 4.2 reports the number of lines of code for EXTFUSE. We also

Table 4.2: Changes made to the existing Linux FUSE framework to support EXTFUSE functionality.

Component	Version	Loc Modified	Loc New
FUSE kernel driver	4.11.0	312	874
FUSE user-space library	3.2.0	23	84
EXTFUSE user-space library	-	-	581

Table 4.3: Metadata can be cached in the kernel using eBPF maps by the user-space daemon and served by kernel extensions.

Metadata	Map Key	Map Value	Caching Operations	Serving Extensions	Invalidation Operations
Inode	<nodeID, name>	fuse_entry_param	`lookup, create, mkdir`	*lookup*	`unlink, rmdir, rename`
Attrs	<nodeID>	fuse_attr_out	`getattr, lookup`	*getattr*	`setattr, unlink, rmdir`
Symlink	<nodeID>	link path	`symlink, readlink`	*readlink*	`unlink`
Dentry	<nodeID>	fuse_dirent	`opendir, readdir`	*readdir*	`releasedir, unlink, rmdir, rename`
XAttrs	<nodeID, label>	xattr value	`open, get(list)xattr`	*get(list)attr*	`close, set(remove)xattr`

modified `libfuse` to allow apps to register kernel extensions.

4.7 Optimizations

Here, we describe a set of optimizations that can be enabled by leveraging custom kernel extensions in EXTFUSE to implement in-kernel handling of file system requests.

4.7.1 Customized in-kernel metadata caching

Metadata operations such as `lookup` and `getattr` are frequently issued, and thus form high sources of latency in FUSE file systems [81]. Unlike VFS caches that are only reactive and fixed in functionality, EXTFUSE can be leveraged to proactively cache metadata replies in the kernel. Kernel extensions can be installed to manage and serve subsequent operations from caches without switching to user space.

Example. To illustrate, let us consider the `lookup` operation. It is the most common operation issued internally by the VFS for serving `open()`, `stat()`, and `unlink()` system calls. Each component of the input path string is searched using `lookup` to fetch the corresponding inode data structure. Figure 4.2 lists code fragment for FUSE daemon handler

```
void handle_lookup(fuse_req_t req, fuse_ino_t pino,
            const char *name) {
    /* lookup or create node @cname parent @pino */
    struct fuse_entry_param e;
    if (find_or_create_node(req, pino, name, &e)) return;
+   lookup_key_t key = {pino, name};
+   lookup_val_t val = {0/*not stale*/, &e};
+   extfuse_insert_shmap(&key, &val); /* cache this entry */
    fuse_reply_entry(req, &e);
}
```

Figure 4.2: FUSE daemon `lookup` handler in user space. With EXTFUSE, lines 6-8 (+) enable caching replies in the kernel.

that serves `lookup` requests in user space (slow path). The FUSE `lookup` API takes two input parameters: the parent node ID and the next path component name. The node ID is a 64-bit integer that uniquely identifies the parent inode. The daemon handler function traverses the parent directory, searching for the child entry corresponding to the next path component. Upon successful search, it populates the `fuse_entry_param` data structure with the node ID and attributes (e.g., size) of the child, and sends it to the FUSE driver, which creates a new inode for the dentry object representing the child entry.

With EXTFUSE, developers could define a `SHashMap` that hosts `fuse_entry_param` replies in the kernel (lines 7-10). A composite key generated from the parent node identifier and the next path component string arguments is used as an index into the map for inserting corresponding replies. Since the map is also accessible to the extensions in the kernel, subsequent requests could be served from the map by installing the EXTFUSE `lookup` extension (fast path). Figure 4.3 lists its code fragment. The extension uses the same composite key as an index into the hash map to search whether the corresponding `fuse_entry_param` entry exists. If a valid entry is found, the reference count (`nlookup`) is incremented and a reply is sent to the FUSE driver.

Similarly, replies from user space daemon for other metadata operations, such as `getattr`, `getxattr`, and `readlink` could be cached using maps and served in the kernel

```
int lookup_extension(extfuse_req_t req, fuse_ino_t pino,
            const char *name) {
    /* lookup in map, bail out if not cached or stale */
    lookup_key_t key = {pino, name};
    lookup_val_t *val = extfuse_lookup_shmap(&key);
    if (!val || atomic_read(&val->stale)) return UPCALL;
    /* EXAMPLE: Android sdcard daemon perm check */
    if (!check_caller_access(pino, name)) return -EACCES;
    /* populate output, incr count (used in FUSE_FORGET) */
    extfuse_reply_entry(req, &val->e);
    atomic_incr(&val->nlookup, 1);
    return SUCCESS;
}
```

Figure 4.3: ExtFUSE `lookup` kernel extension that serves valid cached replies, without incurring any context switches. Customized checks could further be included; Android sdcard daemon permission check is shown as an example.

by respective extensions (Table 4.3). Network FUSE file systems, such as SshFS [86] and Gluster [79] already perform aggressive metadata caching and batching at client to reduce the number of remote calls to the server. SshFS [86], for example, implements its own directory, attribute, and symlink caches. With ExtFUSE, such caches could be implemented in the kernel for further performance gains.

Invalidation. While caching metadata in the kernel reduces the number of context switches to user space, developers must also carefully invalidate replies, as necessary. For example, when a file (or dir) is deleted or renamed, the corresponding cached `lookup` replies must be invalidated. Invalidations can be performed in user space by the relevant request handlers or in the kernel by installing their extensions before new changes are made. However, the former case may introduce race conditions and produce incorrect results because all requests to user space daemon are queued up by the FUSE driver, whereas requests to the extensions are not. Cached `lookup` replies can be invalidated in extensions for `unlink`, `rmdir`, and `rename` operations. Similarly, when attributes or permissions on a file change, cached `getattr` replies can be invalidated in `setattr` extension. Our design ensures race-free invalidation by executing the extensions before forwarding requests to user space daemon

where the changes may be made.

Advantages over VFS caching. As previously mentioned, recent optimizations added to FUSE framework leverage VFS caches to reduce user-kernel context switching. For instance, by specifying non-zero `entry_valid` and `attr_valid` timeout values, dentries and inodes cached by the VFS from previous `lookup` operations could be utilized to serve subsequent `lookup` and `getattr` requests, respectively. However, the VFS offers no control to the user file system over the cached data. For example, if the file system is mounted without the `default_permissions` parameter, VFS caching of inodes introduces a security bug [87]. This is because the cached permissions are only checked for first accessing user. In contrast, with EXTFUSE, developers can define their own metadata caches and install custom code to manage them. For instance, extensions can perform `uid`-based access permission checks before serving requests from the caches to obviate the aforementioned security issue.

Additionally, unlike VFS caches that are only reactive, EXTFUSE enables proactive caching. For example, since a `readdir` request is expected after an `opendir` call, the user-space daemon could proactively cache directory entries in the kernel by inserting them in a BPF map while serving `opendir` requests to reduce transitions to user space. Alternatively, similar to read-ahead optimization, proactive caching of subsequent directory entries could be performed during the first `readdir` call to the user-space daemon. Memory occupied by cached entries could then be freed by the `releasedir` handler in user space that deletes them from the map. Similarly, security labels on a file could be cached during the `open` call to user space and served in the kernel by `getxattr` extensions. Nonetheless, since eBPF maps consume kernel memory, developers must carefully manage caches and limit the number of map entries to keep memory usage under check.

4.7.2 Passthrough I/O for stacking functionality

Many user file systems are stackable with a thin layer of functionality that does not require complex processing in the user-space. For example, LoggedFS [45] filters requests that

must be logged, logs them as needed, and then simply forwards them to the lower file system. User-space union file systems, such as MergerFS [88] determine the backend host file in `open` and redirects I/O requests to it. BindFS [89] mirrors another mount point with custom permissions checks. Android sdcard daemon performs access permission checks and emulates the case-insensitive behavior of FAT only in metadata operations (e.g., `lookup`, `open`, etc.), but forwards data I/O requests directly to the lower file system. For such simple cases, the FUSE API proves to be too low-level and incurs unnecessarily high overhead due to context switching.

With EXTFUSE, `read/write` I/O requests can take the fast path and directly be forwarded to the lower (host) file system without incurring any context-switching if the complex slow-path user-space logic is not needed. Figure 4.4 shows how the user-space daemon can install the lower file descriptor in `InodeMap` while handling `open()` system call for notifying the EXTFUSE driver to store a reference to the lower inode kernel object. With the `custom_filtering_logic(path)` condition, this can be done selectively; for example, if access permission checks pass in Android sdcard daemon. Similarly, BindFS and MergerFS can adopt EXTFUSE to avail passthrough optimization. The `read/write` kernel extensions can check in `InodeMap` to detect whether the target file is setup for passthrough access. If found, EXTFUSE driver can be instructed with a special return code to directly forward the I/O request to the lower file system with the corresponding lower inode object as parameter. Figure 4.5 shows a template `read` extension. Kernel extensions can include additional logic or checks before returning.

4.8 Evaluation

To evaluate EXTFUSE, we answer the following questions:

- **Baseline Performance.** How does an EXTFUSE implementation of a file system perform when compared to its in-kernel and FUSE implementations? (§4.8.1)

```
void handle_open(fuse_req_t req, fuse_ino_t ino,
        const struct fuse_open_in *in) {
    /* file represented by @ino inode num */
    struct fuse_open_out out; char path[PATH_MAX];
    int len, fd = open_file(ino, in->flags, path, &out);
    if (fd > 0 && custom_filtering_logic(path)) {
+       /* install fd in inode map for pasthru */
+       imap_key_t key = out->fh;
+       imap_val_t val = fd; /* lower fd */
+       extfuse_insert_imap(&key, &val);
}   }
```

Figure 4.4: FUSE daemon open handler in user space. With EXTFUSE, lines 7-9 (+) enable passthrough access on the file.

```
int read_extension(extfuse_req_t req, fuse_ino_t ino,
        const struct fuse_read_in *in) {
    /* lookup in inode map, passthrough if exists */
    imap_key_t key = in->fh;
    if (!extfuse_lookup_imap(&key)) return UPCALL;
    log_op(req, ino, FUSE_READ, in, sizeof(*in));
    return PASSTHRU; /* forward req to lower FS */
}
```

Figure 4.5: EXTFUSE `read` kernel extension returns `PASSTHRU` to forward request directly to the lower file system. Custom thin functionality could further be pushed in the kernel.

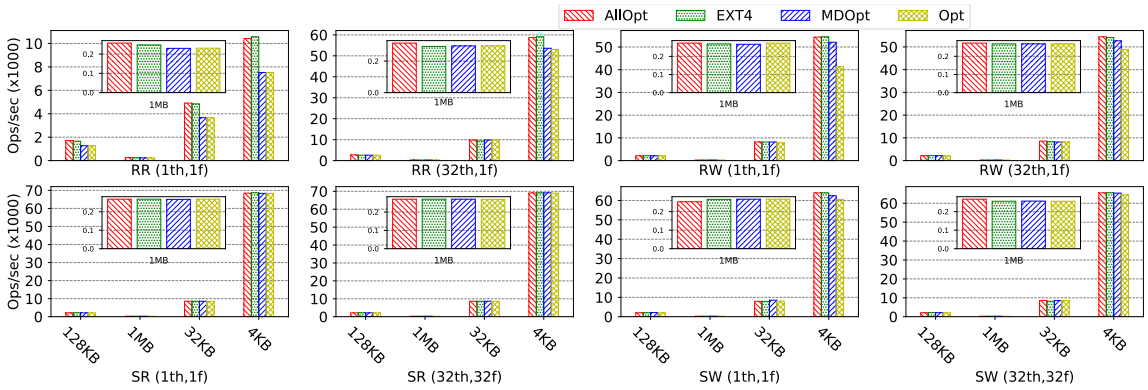


Figure 4.6: Throughput(ops/sec) for EXT4 and FUSE/ExtFUSE Stackfs (w/ xattr) file systems under different configs (Table 4.4) as measured by Random Read(RR)/Write(RW), Sequential Read(SR)/Write(SW) Filebench [90] data micro-workloads with IO Sizes between 4KB-1MB and settings *N*th: *N* threads, *N*f: *N* files. We use the same workloads as in [81].

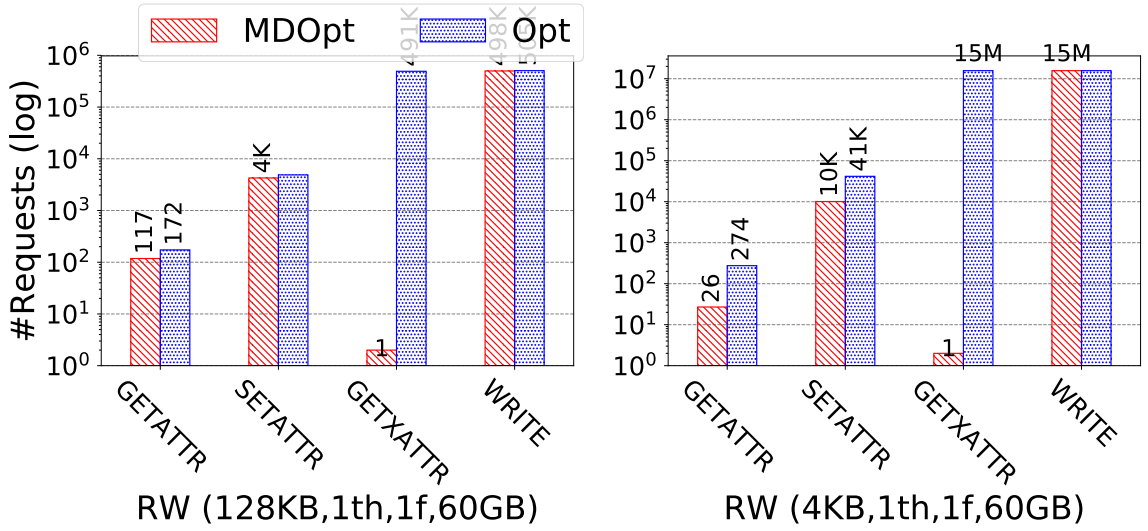


Figure 4.7: Number of file system requests received by the daemon in FUSE/ExtFUSE Stackfs (w/ xattr) under workloads in Figure 4.6. Only a few relevant request types are shown.

- **Use cases.** What kind of existing FUSE file systems can benefit from ExtFUSE and what performance improvements can they expect? (§4.8.2)

4.8.1 Performance

To measure the baseline performance of ExtFUSE, we adopted the simple no-ops (null) stackable FUSE file system called StackFS [81]. This user-space daemon serves all requests by forwarding them to the host (lower) file system. It includes all recent FUSE optimizations

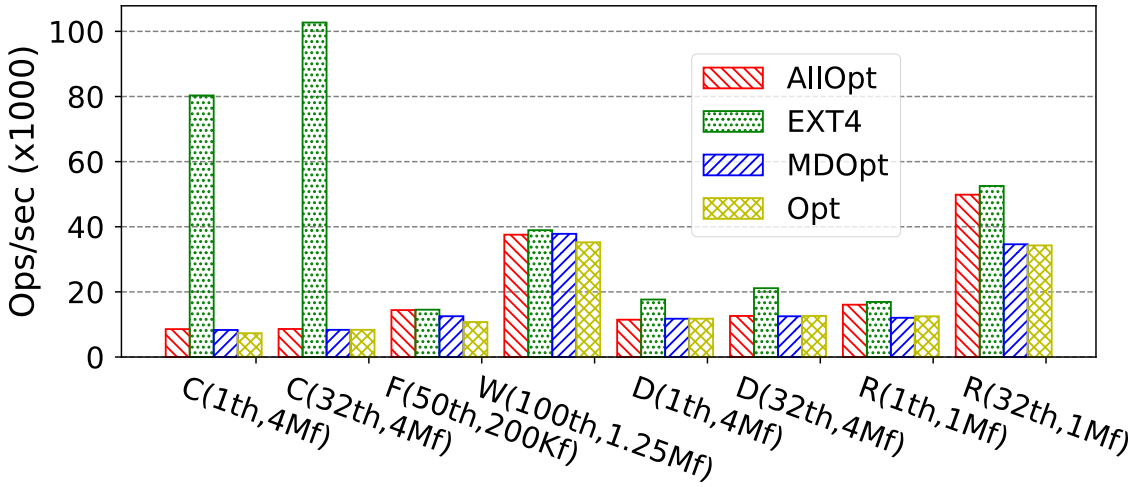


Figure 4.8: Throughput (ops/sec) for EXT4 and FUSE/ExtFUSE Stackfs (w/ xattr) under different configs (Table 4.4) as measured by Filebench [90] Creation(C), Deletion(D), Reading(R) metadata micro-workloads on 4KB files and FileServer(F), WebServer(W) macro-workloads with settings *N*th:*N* threads, *N*f:*N* files.

Table 4.4: Different StackFS configs evaluated.

Config	File System	Optimizations
`Opt` [81]	FUSE	128K Writes, Splice, WBCache, MltThrd
`MDOpt`	ExtFUSE	`Opt` + Caches `lookup`, `attrs`, `xattrs`
`AllOpt`	ExtFUSE	`MDOpt` + Pass `R/W` reqs through host FS

(Table 4.4). We evaluate StackFS under all possible ExtFUSE configs listed in Table 4.4. Each config represents a particular level of performance that could potentially be achieved, for example, by caching metadata in the kernel or directly passing `read/write` requests through the host file system for stacking functionality. To put our results in context, we compare our results with EXT4 and the optimized FUSE implementation of StackFS (`Opt`).

Testbed. We use the same experiments and settings as in [81]. Specifically, we used EXT4 because of its popularity as the host file system and ran benchmarks to evaluate. However, since FUSE performance problems were reported to be more prominent with a faster storage medium, we only carry out our experiments with SSDs. We used a Samsung 850 EVO 250GB SSD installed on an Asus machine with Intel Quad-Core i5-3550 3.3 GHz and

16GB RAM, running Ubuntu 16.04.3. Further, to minimize any variability, we formatted the SSD before each experiment and disabled EXT4 lazy inode initialization. To evaluate file systems that implement `xattr` operations for handling security labels (e.g., in Android), our implementation of `Opt` supports xattrs, and thus differs from the implementation in [81].

Workloads. Our workload consists of Filebench [90] micro and synthetic macro benchmarks to test each config with metadata- and data-heavy operations across a wide range of I/O sizes and parallelism settings. We measure the low-level throughput (ops/sec). Our macro-benchmarks consist of a synthetic file server and web server.

Micro Results. Figure 4.6 shows the results of micro workload under different configs listed in Table 4.4.

Reads. Due to the default 128KB read-ahead feature of FUSE, the sequential read throughput on a single file with a single thread for all I/O sizes and under all StackFS configs remained the same. Multi-threading improved for the sequential read benchmark with 32 threads and 32 files. Only one request was generated per thread for `lookup` and `getattr` operations. Hence, metadata caching in `MDOpt` was not effective. Since FUSE `Opt` performance is already at par with EXT4, the passthrough feature in `AllOpt` was not utilized.

Unlike sequential reads, small random reads could not take advantage of the read-ahead feature of FUSE. Additionally, 4KB reads are not spliced and incur data copying across user-kernel boundary. With 32 threads operating on a single file, the throughput improves due to multi-threading in `Opt`. However, degradation is observed with 4KB reads. `AllOpt` passes all reads through EXT4, and hence offers near-native throughput. In some cases, the performance was slightly better than EXT4. We believe that this minor improvement is due to double caching at the VFS layer. Due to a single request per thread for metadata operations, no improvement was seen with ExtFUSE metadata caching.

Writes. During sequential writes, the 128K big writes and writeback caching in `Opt` allow the FUSE driver to batch small writes (up to 128KB) together in the page cache to

offer a higher throughput. However, random writes are not batched. As a result, more `write` requests are delivered to user space, which negatively affects the throughput. Multiple threads on a single file perform better for requests bigger than 4KB as they are spliced. With ExtFUSE `AllOpt`, all writes are passed through the EXT4 file system to offer improved performance.

Write throughput degrades severely for FUSE file systems that support extended attributes because the VFS issues a `getxattr` request before every `write`. Small I/O requests perform worse as they require more `write`, which generate more `getxattr` requests. `Opt` random writes generated 30x fewer `getxattr` requests for 128KB compared to 4KB writes, resulting in a 23% decrease in the throughput of 4KB writes.

In contrast, `MDOpt` caches the `getxattr` reply in the kernel upon the first call, and serves subsequent `getxattr` requests without incurring further transitions to user space. Figure 4.7 compares the number of requests received by the user-space daemon in `Opt` and `MDOpt`. Caching replies reduced the overhead for 4KB workload to less than 5%. Similar behavior was observed with both sequential writes and random writes.

Macro Results. Figure 4.8 shows the results of macro-workloads and synthetic server workloads emulated using Filebench under various configs. Neither of the ExtFUSE configs offer improvements over FUSE `Opt` under creation and deletion workloads as these metadata-heavy workloads created and deleted a number of files, respectively. This is because no metadata caching could be utilized by `MDOpt`. Similarly, no passthrough writes were utilized with `AllOpt` since 4KB files were created and closed in user space. In contrast, the File and Web server workloads under ExtFUSE utilized both metadata caching and passthrough access features and improved performance. We saw a 47%, 89%, and 100% drop in `lookup`, `getattr`, and `getattr` requests to user space under `MDOpt`, respectively, when configured to cache up to 64K for each type of request. `AllOpt` further enabled passthrough `read/write` requests to offer near native throughput for both macro reads and server workloads.

Real Workload We also evaluated EXTFUSE with two real workloads, namely kernel decompression and compilation of 4.18.0 Linux kernel. We created three separate caches for hosting `lookup`, `getattr`, and `getxattr` replies. Each cache could host up to 64K entries, resulting in allocation of up to a total of 50MB memory when fully populated.

The kernel compilation `make tinyconfig; make -j4` experiment on our test machine (see §4.8) reported a 5.2% drop in compilation time, from 39.74 secs under FUSE `Opt` to 37.68 secs with EXTFUSE `MDOpt`, compared to 30.91 secs with EXT4. This was due to over 75%, 99%, and 100% decrease in `lookup`, `getattr`, and `getxattr` requests to user space, respectively (Figure 4.9). `getxattr` replies were proactively cached while handling `open` requests; thus, no transitions to user space were observed for serving xattr requests. With EXTFUSE `AllOpt`, the compilation time further dropped to 33.64 secs because of 100% reduction in `read` and `write` requests to user space.

In contrast, the kernel decompression `tar xf` experiment reported a 6.35% drop in the completion time, from 11.02 secs under FUSE `Opt` to 10.32 secs with EXTFUSE `MDOpt`, compared to 5.27 secs with EXT4. With EXTFUSE `AllOpt`, the decompression time further dropped to 8.67 secs due to 100% reduction in `read` and `write` requests to user space, as shown in Figure 4.9. Nevertheless, reducing the number of cached entries for metadata requests to 4K resulted in a decompression time of 10.87 secs (25.3% increase) due to 3,555 more `getattr` requests to user space. This suggests that developers must efficiently manage caches.

4.8.2 Use cases

We ported four real-world stackable FUSE file systems, namely LoggedFS, Android sdcard daemon, MergerFS, and BindFS to EXTFUSE and enabled both metadata caching §4.7.1 and passthrough I/O §4.7.2 optimizations.

As EXTFUSE allows file systems to retain their existing FUSE daemon code as the default slow path, adopting EXTFUSE for real-world file systems is easy. On average, we

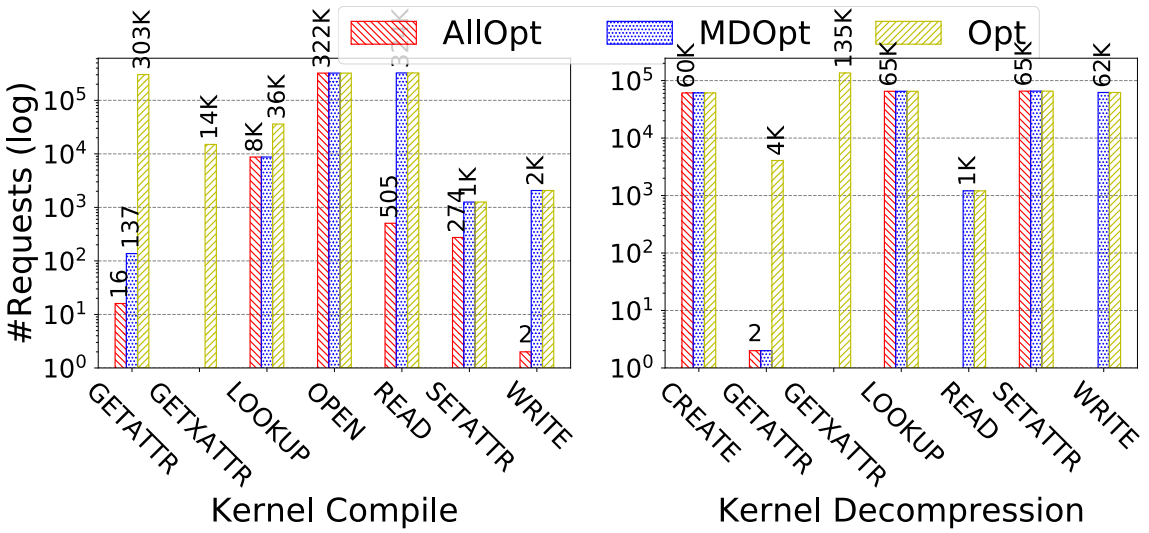


Figure 4.9: Linux kernel 4.18.0 untar (decompress) and compilation time taken with StackFS under FUSE and ExtFUSE settings. Number of metadata and I/O requests are reduced with ExtFUSE.

Table 4.5: Lines of code (Loc) of kernel extensions required to adopt ExtFUSE for existing FUSE file systems. We added support for metadata caching as well as R/W passthrough.

File System	Functionality	Ext Loc
StackFS [81]	No-ops File System	664
BindFS [89]	Mirroring File System	792
Android sdcard [77]	Perm checks & FAT Emu	928
MergerFS [88]	Union File System	686
LoggedFS [45]	Logging File System	748

made less than 100 lines of changes to the existing FUSE code to invoke ExtFUSE helper library functions for manipulating kernel extensions, including maps. We added kernel extensions to support metadata caching as well as I/O passthrough. Overall, it required fewer than 1000 lines of new code in the kernel Table 4.5. We now present detailed evaluation of Android sdcard daemon and LoggedFS to present an idea on expected performance improvements.

Android sdcard daemon. Starting version 3.0, Android introduced the support for FUSE to allow a large part of internal storage (e.g., `/data/media`) to be mounted as external FUSE-managed storage (called `/sdcard`). Being large in size, `/sdcard` hosts user data,

such as videos and photos as well as any auxiliary Opaque Binary Blobs (OBB) needed by Android apps. The FUSE daemon enforces permission checks in metadata operations (e.g.,`lookup`, etc.) on files under `/sdcard` to enable multi-user support and emulates case-insensitive FAT functionality on the host (e.g., EXT4) file system. OBB files are compressed archives and typically used by games to host multiple small binaries (e.g. shade rs, textures) and multimedia objects (e.g. images, etc.).

However, FUSE incurs high runtime performance overhead. For instance, accessing OBB archive content through the FUSE layer leads to high launch latency and CPU utilization for gaming apps. Therefore, Android version 7.0 replaced sdcard daemon with with an in-kernel file system called `SDCardFS` [77] to manage external storage. It is a wrapper (thin) stackable file system based on WrapFS [85] that enforces permission checks and performs FAT emulation in the kernel. As such, it imposes little to no overhead compared to its user-space implementation. Nevertheless, it introduces security risks and maintenance costs [91].

We ported Android sdcard FUSE daemon to ExtFUSE framework. First, we leverage eBPF kernel helper functions to push metadata checks into the kernel. For example, we embed access permission check (Figure 4.10) in `lookup` kernel extension to validate access before serving `lookup` replies from the cache (Figure 4.3). Similar permission checks are performed in the kernel to validate accesses to files under `/sdcard` before serving cached `getattr` requests . We also enabled passthrough on `read/write` using `InodeMap`.

We evaluated its performance on a 1GB RAM HiKey620 board [92] with two popular game apps containing OBB files of different sizes. Our results show that under `AllOpt` passthrough mode the app launch latency and the corresponding peak CPU consumption reduces by over 90% and 80%, respectively. Furthermore, we found that the larger the OBB file, the more penalty is incurred by FUSE due to many more small files in the OBB archive.

LoggedFS is a FUSE-based stackable user-space file system. It logs every file system operation for monitoring purposes. By default it writes to syslog buffer and logs all

Table 4.6: App launch latency and peak CPU consumption of `sdcard` daemon under default (D), and passthrough (P) settings on Android for two popular games. In passthrough mode, the FUSE driver never forwards `read/write` requests to user space, but always passes them through the host (EXT4) file system. See Table 4.4 for config details.

App Stats		CPU (%)		Latency (ms)	
Name	**OBB Size**	**D**	**P**	**D**	**P**
Disney Palace Pets 5.1	374MB	20	2.9	2235	1766
Dead Effect 4	1.1GB	20.5	3.2	8895	4579

```
bool check_caller_access_to_name(int64_t key, const char *name) {
    /* define a shmap for hosting permissions */
    int *val = extfuse_lookup_shmap(&key);
    /* Always block security-sensitive files at root */
    if (!val || *val == PERM_ROOT) return false;
    /* special reserved files */
    if (!strncasecmp(name, "autorun.inf", 11) ||
        !strncasecmp(name, ".android_secure", 15) ||
        !strncasecmp(name, "android_secure", 14))
        return false;
    return true;
}
```

Figure 4.10: Android `sdcard` permission checks ExtFUSE code.

operations (e.g., open, read, etc.). However, it can be configured to write to a file or log selectively. Despite being a simple file system, it has a very important use case. Unlike existing monitoring mechanisms (e.g., Inotify [43]) that suffer from a host of limitations [44], LoggedFS can reliably post all file system events. Various apps, such as file system indexers, backup tools, Cloud storage clients such as Dropbox, integrity checkers, and antivirus software subscribe to file system events for efficiently tracking modifications to files.

We ported LoggedFS to ExtFUSE framework. Figure 4.11 shows the common logging code that is called from various extensions, which serve requests in the kernel (e.g., `read` extension Figure 4.5). To evaluate its performance, we ran the FileServer macro benchmark with synthetic a workload of 200,000 files and 50 threads from Filebench suite. We found

over 9% improvement in throughput under `MDOpt` compared to FUSE `Opt` due to 53%, 99%, and 100% fewer `lookup`, `getattr`, and `getxattr` requests to user space, respectively. Figure 4.12 shows the results. `AllOpt` reported an additional 20% improvement by directly forwarding all `read/write` requests to the host file system, offering near-native throughput.

```
void log_op(extfuse_req_t req, fuse_ino_t ino,
        int op, const void *arg, int arglen) {
    struct data { /* log record */
        u32 op; u32 pid; u64 ts; u64 ino; char data[MAXLEN];};
    /* example filter: only whitelisted UIDs in map */
    u16 uid = bpf_get_current_uid_gid();
    if (!extfuse_lookup_shmap(uid_wlist, &uid)) return;
    /* log opcode, timestamp(ns) and requesting process */
    data.opcode = op; data.ts  = bpf_ktime_get_ns();
    data.pid = bpf_get_current_pid_tgid(); data.ino = ino;
    memcpy(data.data, arg, arglen);
    /* submit to per-cpu mmap'd ring buffer */
    u32 key = bpf_get_smp_processor_id();
    bpf_perf_event_output(req, &buf, &key, &data, sizeof(data));
}
```

Figure 4.11: LoggedFS kernel extension that logs requests.

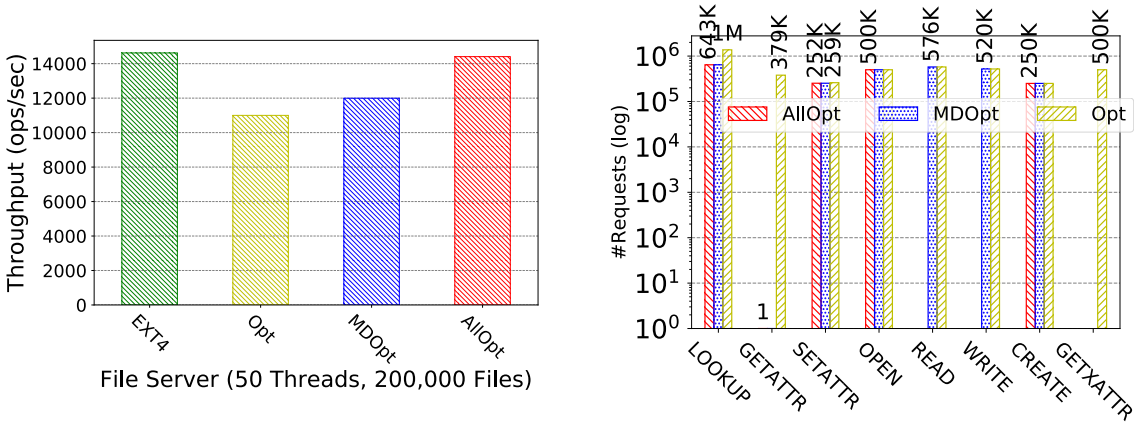


Figure 4.12: LoggedFS performance measured by Filebench FileServer benchmark under EXT4, FUSE, and ExtFUSE. Fewer metadata and I/O requests were delivered to user space with ExtFUSE.

CHAPTER 5
CONTEXT-AWARE STORAGE ARCHITECTURE

In this chapter, we first present a motivational use case for our proposed system, then introduce our design goals, challenges in meeting those goals, and mechanisms adopted to address the challenges.

5.1 Use case

We would like to illustrate the functionality of proposed smart storage service for smart mobile devices using the following example scenario. Sarah, a biologist, uses her personal smartphone to gain access to her work-related data for better productivity. The same device also hosts her health and fitness apps, social networking apps, favorite music files, and pictures. Sarah listens to music every morning while working out and watches movies on her device whenever she gets a chance.

Sarah has to present at a conference, for which she is flying out soon. Fortunately, Sarah has a smart storage system on her device that accesses her calendar to know that she has a flight to catch and makes sure her "flight-mode" data, including her favorite movies and are available locally on the device. The night before the scheduled conference event, when Sarah puts the device on charging, the system automatically offloads movies and downloads work-related documents for the next day.

She is also an avid photographer; she frequently takes pictures and shares with her friends on social networking. After the conference, as Sarah goes out for sightseeing, the system, based on the location (e.g. a popular tourist place), proactively makes space for more pictures. Next morning, when she is about to go for a run, the system makes sure that her workout music files are prefetched.

As the device hosts work data as well, the phone may run out of storage space anytime.

However, a smart storage system on her device eschews this problem. Once she reaches her workplace, where she is less likely to access personal videos and pictures, the system automatically reclaims storage space occupied by her personal media files to accommodate work-related data.

5.2 Goals

Drawing upon the findings from our user study, the key goal of ANODYNE is to perform proactive context-sensitive quota-based management to make more productive use of storage resources in anticipation of a foreseen or unexpected demand. Below we identify specific goals that will enable our system to function efficiently.

Default application transparency and portability. Besides offering native performance, the system must be compatible with existing interfaces, and must not require any mandatory changes to the apps, or the underlying device storage. Nevertheless, it must also provide optional APIs (hooks) for apps that want to implement customized functionality for storage management events.

Hierarchical storage management. Users already rely on Cloud storage to back up data and free storage space on their devices [11]. However, existing services, such as Dropbox [13] and Google Drive [15] are available as third-party apps, and are not integrated with the underlying storage. This results in a fragmented view of overall storage resources, ruling out a system-level automated management. Our goal is to enable integration of Cloud storage and perform automated hierarchical management when needed.

5.3 Challenges

While enforcing context-sensitive quota offers an attractive solution to the problem of constrained storage on smart mobile devices, a poorly designed system could hurt performance, waste resources, and severely degrade user experience. Hence, a host of deployment challenges need to be addressed.

Latency sensitivity. First, mobile devices are used for short periods of time, typically a few minutes or even seconds. Thus, users expect fast response, making these devices highly latency-sensitive. User interactions on mobile devices are short, typically lasting a few minutes or even seconds [93, 94]. Thus, fast response is highly desirable, making these devices particularly latency-sensitive. Apps with large latencies are at a risk of losing users [95] or being killed by the operating system [96]. Reconstructing data on demand by computation (e.g., decompression) or fetching from the Cloud, may incur significant latency and degrade user experience [95]. Thus, accurately predicting the storage needs based on active user context is very important.

Storage Reclamation. Apps operate on diverse set of data, from unstructured binary files to structured SQLite database and XML key-value objects across multiple dirs §2. Determining what and how much data to reclaim or hoard can be tricky and have serious user-experience implications. For instance, while cached content can be safely deleted and regenerated when needed (e.g., by computation or web access), blind removal of app data, such as active game state can cause user frustration. Therefore, the system must carefully choose what reclamation technique to apply to what kind of data.

Resource constraints. Mobile devices are resource constrained; they work with limited storage, battery, and CPU. Users also have monthly-budgeted cellular data. Infrequently accessed data objects are good candidates for reclamation, but their reconstruction may incur high network bandwidth, cellular data, battery, and CPU resources. Therefore, the system must carefully balance resource trade off. Similarly, continuous monitoring of device to build storage working-set profiles for context-sensitive management may also incur high resource usage if not done efficiently.

5.4 Storage Reclamation

ANODYNE applies multiple reclamation techniques, such as deletion, compression, content-adaptation, and Cloud-backed hierarchical management to recover storage space when

the consumption on the device is low. Since blind removal of data can lead to poor user-experience as discussed in §5.3, the system carefully selects a reclamation technique depending on the data type. Table 5.1 reports the techniques we use for various data types.

App Code. ANODYNE uses deduplication, deletion, content-adaptation, and compression to reclaim unnecessary storage space occupied by app binaries. For instance, deduplication saves space consumed by common code across apps from the same vendor or using the same SDKs §3.4.

As reported in §3.4, apps expand to further consume at least 1.5x of storage space upon fresh installation. This is mainly due to creation of precompiled OAT executable for all DEX Java class and extraction of all native libraries from the APK to offer high runtime performance. The assumption behind creating performance-optimized OAT files is that all app features are equally important to the user at all times, which is not true as shown by our user study §3.5.2. Therefore, ANODYNE performs context-aware content adaptation to reclaim space occupied by unwanted Java executables. Specifically, we use selective AOT compilation on *only a subset* of DEX classes that are predicted to be used. This reduces the storage consumption of OAT file as Java classes that are not used (e.g., in-app purchases) will not be precompiled. If the app is predicted to not be used until the following day, then ANODYNE temporarily deletes its OAT file, freeing up at least 60% of the storage space. In case of misprediction, the system falls back to runtime interpretation of DEX bytecode, incurring additional latency and energy consumption. Nevertheless, ANODYNE could opportunistically regenerate the OAT file by performing *dex2oat* on DEX files when the device is charged overnight.

Similarly, not all native libraries are actively used. To recover space, ANODYNE can safely delete inactive libraries since they are a part of the compressed APK file, which is kept on the device until the app is uninstalled. Under misprediction, the system decompresses them on the fly at the cost of higher latency and energy consumption.

Finally, ANODYNE also reclaims space occupied by files that are incompatible on the

Table 5.1: Multiple techniques are employed by ANODYNE to reclaim storage space as some files host critical and stateful data, whereas others could be regenerated on-demand.

Type	Examples	Critial	Auto-regenerated	Reclamation Technique(s)
App Code	OAT, Libs	All	All	Content Adapt, Dedup, Compression
App Cache	Web cache	None	All	LRU Deletion
App State	DBs, Prefs	All	None	Compression, HM
App Data	Files	Some	Some	Compression, Deletion, HM
User Data	Pics, Docs	All	None	LRU Hierarchical Management (HM)

device (e.g., x86 and MIPS code on an ARMv7 device).

App Caches. Apps frequently cache or prefetch data from the Cloud to offer low latency, at the cost of additional resource (e.g., storage, cellular data, battery, etc.) consumption. Cached data is temporary, and can be regenerated by either computation on the device or fetching contents from a Cloud server. Nevertheless, deleting cached working set of an app may incur high latency and resource consumption. Therefore, ANODYNE recovers storage space occupied by temporary files in *cache* by deleting them in a Least Recently Used (LRU) fashion since it captures the inactive data.

To identify what data is temporary for an app, we tap into the Android framework that requires apps to store a particular data type in its designated app dir. For example, temporary data (e.g., web cache) is placed in *cache* and stateful critical data (e.g., user preferences, etc.) in *databases* or *shared_prefs* dir (see Table 2.3). Apps that do not honor these requirements may lead to poor user experience. For instance, the *storaged* daemon in Android can delete files in *cache* dir without any prior notification to recover space when the total storage consumption reaches 90% [97]. Users can also manually delete *cache* contents anytime. As such, storing stateful data in *cache* dir that is critical to the correct functioning of the app may result in loss of functionality or even crash the app if such files are deleted. Therefore, we believe that developers have a high incentive to develop apps that honor such requirements.

App State. *databases* and *shared_prefs* dirs contain stateful persistent data (e.g., user preferences). Such data is highly critical for native user experience; it cannot be regenerated without additional user input. Thus, app state must be persisted and not blindly deleted.

App Data. In contrast, not all content in *files* can be easily categorized as temporary or critical. As reported in §3.5, some apps extract *assets* from its APK and place them in *files* dir to offer higher performance. ANODYNE can safely delete them to free up space when needed, and regenerate by extracting them from the APK. However, some apps also freely hoard non-critical data (e.g., analytics, advertisements, logs) in *files* (see §3.5). As such, attempting to automatically label every file as critical or temporary could either lead to errors or wastage of storage. Therefore, we set a hard limit of 25MB (compressed) on the overall persistent storage quota for hosting data of contextually inactive apps. The limit is based on the backup size limits imposed by Android framework while performing auto-backup of apps, which is enough to enable seamless and transparent inter-device transfer of critical data for users during device upgrade [98].

ANODYNE provides an optional *enforcequota* API for apps that wish to temporarily host persistent data beyond the quota limits. When invoked, the app handler must reclaim data to yield final consumption within limits. Failure to do so will result in deletion of all persistent data, including any user data (e.g., user credentials), which is equivalent to fresh app start behavior (e.g., re-enter credentials).

ANODYNE uses compression or Cloud-backed hierarchical management to reclaim space occupied by persistent data, depending upon the storage demand. The latter results in auto-backup [98] of an app, where all its persistent data is compressed and stored in Cloud. In case of misprediction, data is decompressed on the fly and restored on the device from the Cloud. Network connectivity failure during the latter will result in fresh app start similar to new installation (e.g., no saved account info).

Some apps may save active state on Cloud servers (e.g., games). Therefore, ANODYNE provides optional *reclaim* and *regenerate* hooks for apps. These hooks allow them to be notified of storage management events, and manage their data. Reclaim hooks are called immediately before any data is reclaimed to allow apps to back up data on their Cloud servers. Whereas, regenerate hooks are called when the data needs to be reconstructed. It

allows apps to regenerate data (e.g., download the latest version of a game state from the Cloud). This is inspired by *background refresh* in Android. Therefore, when ANODYNE predicts usage of an app, it calls the reconstruction hook to regenerate necessary files.

User Data. To free up space occupied by user data, such as pictures, videos, and documents, ANODYNE performs hierarchical management using existing popular Cloud storage services, such as Google Drive and Dropbox. Users already use Cloud storage to backup their personal data. However, ANODYNE integrates Cloud storage into the file system for centralized management of all storage resources.

5.5 Continuous System Profiling

To enable context-aware management, ANODYNE continuously collects file system traces along with contextual data.

Contextual data contain timestamped values expressing multiple attributes, such as current app and features (Android Activities), location, network connectivity, and user physical activity. Table 3.5 presents a list of all contextual attributes we use. To minimize resource overhead imposed by continuous monitoring, ANODYNE subscribes to updates from various stateful events on the device (e.g., location updates, etc.) and logs them as they occur as opposed to periodically polling the sensors on the device. It further leverages rich Cloud-augmented services on smart devices to receive updates to device location and user's current and planned activities. For example, Google *Activity Recognition* API on Android devices or *CMMotionActivity* API on iOS devices can be used to ascertain if the user is walking or driving. Similarly, Google location and calendar services can be used to report user's current location and upcoming events. Such services obtain authorization from users to gain access to their sensitive data and are already optimized for battery consumption. A number of apps already rely on such services.

ANODYNE system also tracks and logs file system calls to identify what files are accessed under what context. File system traces include timestamped logs of file system operations

performed (e.g., read, write, etc.), access paths, and relevant metadata. Traces are only collected for file system events generated by apps the user interacts with.

Data collected is stored in a highly compressed manner. We use *protobuf* format to store and transfer data. Figure 5.3 shows an excerpt. This not only enables continuous system profiling with low storage footprint, but also offers flexibility to both hold onto the data under disconnections or immediately send it to the Cloud agent incurring negligible cellular data. To protect user privacy, data could be anonymized. For example, device id, apps names, and file paths could be obfuscated using hashes. Depending upon the learning algorithm data could be batched and uploaded once a day or every few hours.

5.6 Architecture and Workflow

Figure 5.1 shows the architecture of mobile storage service. It consists of the following components: a mobile *storage service* and a Cloud-hosted *prediction agent*. In this section, we discuss the architecture of mobile storage service and provide details about the Cloud agent as pertinent to the discussion.

The mobile storage service is a privileged system service that is responsible for the core storage management services of the system, such as tracking storage usage, data reclamation, and regeneration. To achieve app-transparency and portability, the service presents itself as a stackable file system, i.e. it uses the lower (host) file system to perform I/O and store data, while adding a layer atop with storage elasticity (i.e., reclaim space when needed) and management as the first-class goals. The layered design enables compatibility with existing OS interfaces, requires no change to the storage layout, and decouples the management functionality from the host file system implementation.

Workflow. To understand how data tracing and reconstruction process, let us look at an example of a file system request in detail. As shown in Figure 5.1, when an app makes a file system request (e.g., `read`, `write`, etc.) ❶ , the request is forwarded to the overlay file system ❷ . The file system then first checks the state of the requested file (or

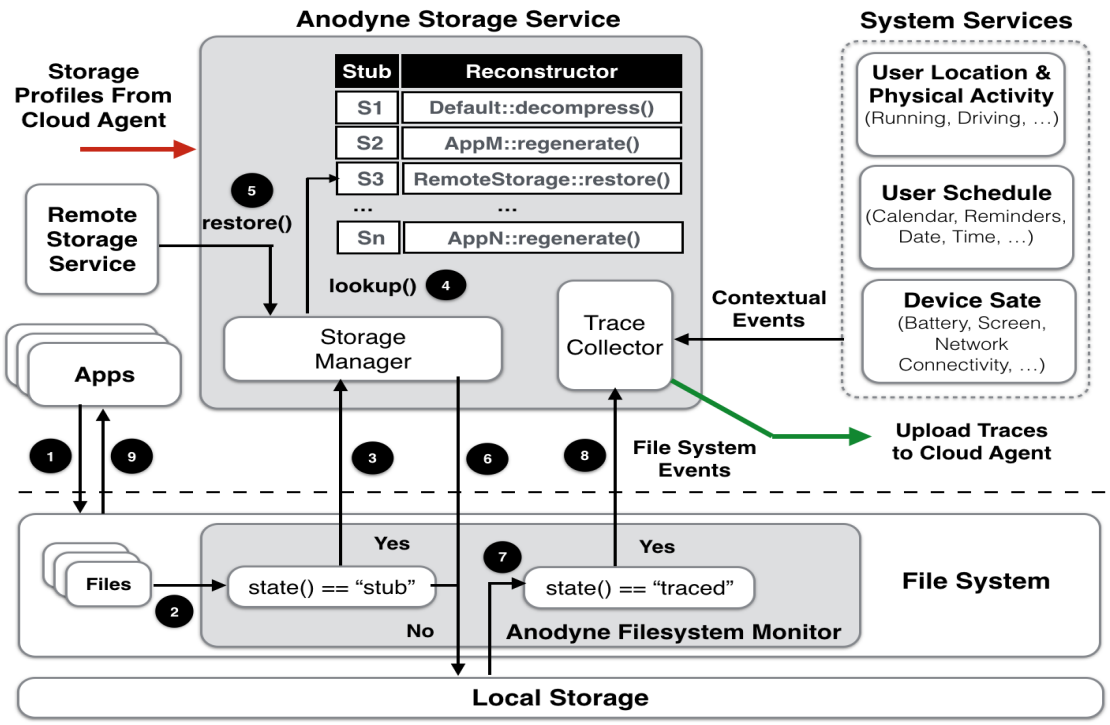


Figure 5.1: ANODYNE Mobile Client Storage Service Architecture This figure shows the ANODYNE mobile storage service architecture. The components introduced are highlighted in grey.

directory) to determine if it is a *stub*; that is, if its storage space has been reclaimed, and data reconstruction is needed. In case of a stub, the request is forwarded to the *storage manager* ❸ , which looks up ❹ and invokes ❺ the registered data reconstruction handler.

5.7 Context-sensitive Storage Profiles

Our storage prediction engine is hosted on a remote Cloud server. It analyzes data collected by mobile service running on users' devices, performs feature extraction and is responsible for building storage profiles and predicting users' storage needs for each profile.

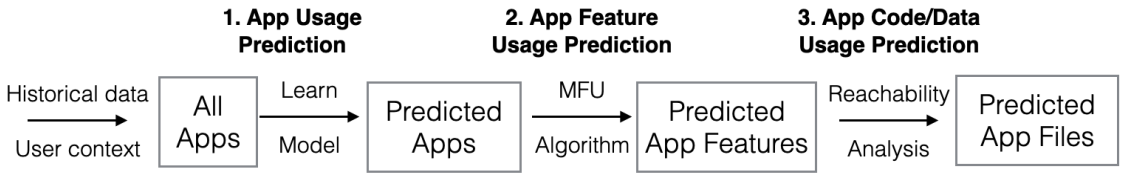


Figure 5.2: Context-sensitive storage working set for a user is predicted by first determining what apps are likely to be used given user active context and historical app usage data.

App usage. To learn context-sensitive storage working set of a user, first the app usage

behavior of the user is learnt; i.e., apps that are likely to be used given the context. We formulate it as a multi-label prediction problem: given a list of installed apps and their historical usage data, predict the apps likely to be used in the next time interval. Inspired by several prior works [99, 100, 101, 102, 103], we also rely on proven contextual attributes, such as device location, time and day of the week, and last "n" apps to model future app usage. We include additional features, such as phone charging status and physical activity of the user to improve prediction.

Apps that are used once or twice daily or weekly (e.g., parking) are perfect candidates for reclamation under high storage pressure. Such apps are closely related to user context/schedule (e.g., parking or fitness apps in the morning, email at work, alarm at night); thus, lend an opportunity to reclaim space until next use. Accurately predicting infrequently used apps (e.g., parking) is critical for efficient storage management. However, the infrequency of their usage causes class imbalance in the dataset, posing a accuracy challenge. As a workaround, we filter out very frequently used apps (e.g., Facebook, WhatsApp) and focus on tracking infrequently used apps in our prediction model.

Furthermore, predictive storage management on a smart mobile device can be performed in multiple ways. First, is to predict storage needs per-day basis. Since users are more likely to charge their devices once in a day (potentially overnight at home, with WiFi connectivity), reconstructing files to prepare the use of device for the next day comes at almost no overhead in terms of budgeted resources. (e.g., battery, cellular data, etc.) Second, even within the day, storage needs are predictable at per-hour basis. For instance, our user study shows the use of Yelp app to search for restaurants is a more likely event around lunch or dinner time. However, depending upon the type of data, the latter scheme can incur high usage of resources, but provide native runtime performance. We use a hybrid approach; ANODYNE manages storage on per-day basis and also opportunistically multiple times during the day.

Data Preprocessing. We prepare the collected data as spatio-temporal sequence of apps used everyday. Every data-point as a 4-tuple of the form `(T,D,L,P,S,A)`, representing time

```
Network-Wifi-Anodyne {
"Action":"android.net.wifi.STATE_CHANGE",
    "WifiInfo":{
        "SupplicantState":"COMPLETED",
        "SSID":"XXX",
        "BSSID":"XXX",
        "MAC":"XXX",
    }
    "timestamp":XXX
}

Context-Activity-Anodyne {
    "Action":"UserActivity",
    "Vehicle":67,
    "Bicycle":15,
    "Unknown":15,
    "Still":3,
    "timestamp":TTT
}
```

```
Context-Location-Anodyne {
    "Action":"android.location.LOCATION_CHANGED",
"Location":{
        "Provider":"fused",
        "Time":XXX,
        "ElapsedRealtimeNanos":XXX,
        "Latitude":XXX,
        "Longitude":XXX,
        "HasAltitude":false,
        "Altitude":0,
        "HasSpeed":false,
        "Speed":0,
        "HasBearing":false,
        "Bearing":0,
        "HasAccuracy":true,
        "Accuracy":25.189,
        "IsFromMockProvider":false
    },
    "timestamp":XXX
}
```


Figure 5.3: An excerpt from data traces collected from a particular user. Sensitive fields have been redacted.

of the day (T), day of the week (D), device location (L), user physical activity (P), phone status (S), and app (A) usage. Since time is continuous, we feed in segmented temporal information to the model. Specifically, we divide the day into eight 3-hour *phases*.

Like [99, 102], we also convert location data into clusters of significant places that the user visits. We use DBSCAN spatial clustering algorithm [104] to first group locations within the radius of 500 meters together. Each of these clusters are annotated with a set of features, such as their frequency, time, and duration of visits, WiFi/cellular info, and nearby places (e.g., parks) using OpenStreet API [105]. We leverage OpenStreet API [105] to obtain nearby places and points-of-interests (e.g., parks, malls) around the cluster centroid. During visit to an unseen location, the place type helps the system identify what files will be accessed. Based on the features, we classify a cluster into various categories (e.g., home, work) using multiple common heuristics [106].

Model. Our input contains discrete codes for phase changes during the day (e.g., morning, night, etc.), change in location category (e.g., home, work, etc.), physical activity of the user (e.g., driving, exercise, etc.), and phone status (e.g., charging, silent, etc.). We use Support Vector Machine (SVC) for training and predicting both the set of next-day as well as next-hour apps that are likely to be used. Our results suggest that SVC is the best model,

Table 5.2: Contextual data collected.

Attribute	Android Service	Context Retrieved
App Usage	ActivityManager	App, Activity Name
Timestamp	System Time	Time, Day, Month, Year
Location	LocationServices	Location Updates
Wireless	WifiManager	Wifi SSID
UserActivity	ActivityRecognition	e.g., walking, etc.

because it have low miss-prediction ratio and over-prediction ratio §5.10.3.

Storage working set. For each predicted app, the agent analyzes the app historical usage data to determine what app features (e.g., Android Activities) are likely to be used next and assigns a score to each. We found that Most Frequently Used (MFU) algorithm works the best for most cases across users. We further apply multiple heuristics to improve the feature score. For instance, some categories of apps (e.g., games, ebooks) have sequential usage behavior (e.g., games levels, book chapters). In such cases, we assign higher score to the next consecutive feature(s) in the usage sequence. Similarly, apps with in-app purchases of features, we assign a higher score to the purchased feature. Such scores can be further improved by considering the history of feature usage across apps. We leave this as future work.

The agent then performs static reachability analysis of app code to determine what Java classes, native libraries, and persistent data files (e.g., `databases`, `shared_prefs`, etc.) will be loaded by each predicted app Activity (feature). Limitations of static analysis due to code obfuscation can be overcome by using JIT profile data. The timestamps on past file system traces are used to further correlate file accesses with app usage. Similarly, accesses to files in `cache` dir are analyzed to track temporary working set of an app and enforce LRU deletion policy. Each file is finally assigned a score that represents its likelihood of being accessed under a given user context.

5.8 Storage Management

Optimizer. ANODYNE includes an optimizer module to decide what and how much system resources (e.g., battery) must be traded to gain storage. Based on the context-sensitive working set profiles, it determines what app files could be will be needed next.

Let $F = \{f_1, f_2, ..., f_n\}$ denote the set of files stored on user's device, and C be the user's spatio-temporal context. For each file $f_i \in F$, let s_i be the size of storage space that can be reclaimed, and o_i be the overhead of reconstructing it. If S is the minimum amount of storage space that the system needs to reclaim, the problem of context-sensitive storage management is to find a subset of files that can be reclaimed to achieve overall storage savings of size S, such that the overhead of reconstructing them under C is minimized. Specifically, the problem is formally defined as follows:

$$\sum_{i=1}^{n} s_i x_i \geqslant S,\ s.t.\ min\ \sum_{i=1}^{n} o_i x_i,\ where\ x_i \in \{0, 1\}$$

Because storage space occupied by an object could be reclaimed and later reconstructed in multiple ways, different approaches will have different resource consumption For example, space occupied by app persistent data could be reclaimed by either compression or Cloud-backed HM (see §5.4). In the example above, decompression consumes CPU and battery resources, whereas reconstructing data by fetching content from the Cloud consumes battery, CPU as well as network resources. Therefore, we consider multiple reconstruction approaches.

Reconstruction overhead o_i is calculated as,

$$o_i = (1 - Score(f_i))\ (Lat(f_i) + Pwr(f_i) + Net(f_i))$$

Where $Lat(f_i)$, $Pwr(f_i)$, and $Net(f_i)$ are performance latency, battery overhead, and network data consumed due to on-demand reconstruction of file f_i when mispredicted. This

problem can be formulated as the 0-1 KNAPSACK combinatorial optimization problem, which is N P - complete.

However, the $O(nW)$ running time where $W = \sum_{i=1}^{n} s_i - S$ is the maximum space for remaining files, is prohibitive on systems with a large number of files. Therefore, we use greedy approach with reconstruction overhead. In the approach, we consider a set A of the least recently used K items of the LRU queue, doubling the value of K as required to obtain until at least S bytes of space can be recovered. We sort A by the ratio of s_i and greedily select high ratio files from o_i set A until we reach or exceed the S threshold.

If the prediction is incorrect, the file will be reconstructed on demand. ANODYNE therefore will have to predict on-demand reconstruction overhead in the future. Since latency depends on the system load at the time of the access, in our current implementation, we make the simplifying assumption that the system load is relatively stationary over time, and generate latency estimates by averaging the conditions observed in the past for the device as a reasonable approximation of future conditions. ANODYNE tracks average energy consumption, access latency, and network bandwidth to calculate o_i. However, predicted files may never consumed, so ANODYNE must also consider the prediction accuracy of the system when making overhead estimates. Accuracy is maintained by ANODYNE as the number of predicted files which were consumed by the system divided by the total number of predicted files.

5.9 Implementation

To evaluate the design implications of our proposed ANODYNE storage architecture, we have implemented its working prototype for Android Marshmallow (v6.0.1, API 23) on LG Nexus 5 device. In this chapter, we describe our implementation details.

5.9.1 File System

The storage management and elasticity layer in ANODYNE is implemented as a stackable EXTFUSE user file system §4. Being in user space, it makes is easier to implement multiple storage reclamation and data reconstruction techniques with varying degrees of complexity (e.g., deduplication, compression, hierarchical management) This also allow reuse of existing third-party libraries (e.g., zlib, Cloud storage SDKs). To avoid unnecessary context switches to user space, we implement ExtFUSE kernel extensions that are serve file requests in the kernel if no reconstruction is required (i.e., file is not a stub).

For example, read requests to compressed *stubs* are forwarded to the user-space. Remaining requests are simply forwarded to the lower (host) the file system, which are served at native speed. We use eBPF [84] hash maps shared between the user-space daemon and extensions in the kernel to identify mark reclaimed files. Similarly, inode maps [107] is used to directly forward I/O to the target file in case the requested file is deduplicated. Figure 5.4 shows the kernel extension code.

5.9.2 Storage service

The storage service is the core part of ANODYNE. It is responsible for a number of tasks, such as tracking storage accesses and user context as well as reclaiming and reconstructing data. It is privileged system service implemented as a user-space daemon that is forked from the `init` process as the system boots. The service starts with `CAP_SYS_ADMIN` access for full functionality, but drops system privileges to user-level after mounting the ANODYNE file system. It is about 3800 lines of code written in C/C++ and Java, with additional dependent libraries.

To reclaim space occupied by OAT files, we leverage AOT compilation on Android. As mentioned previously, it is configurable. We leverage into configurable behavior of AOT to apply quota on app code. The AOT compilation is accomplished using *dex2oat* tool. On first installation, there is no oat file generated . So the app is run with JIT, which profiles

and logs the hot code. We use Soot framework [108] to perform static reachability analysis on Java DEX classes.

We build a custom app based on `Google Drive Android API` and use it as the designated Cloud storage platform app to perform hierarchical storage management. Reclaimed user data files are uploaded to Google Drive and a `link` to its thumbnail is created.

```
void trace_io(extfuse_req_t req, fuse_ino_t ino,
        int op, const void *arg, int arglen) {
    struct io_record rec; /* log record */
    /* log I/O operation with timestamp(ns) */
    rec.opcode = op; rec.ino = ino;
    rec.ts  = bpf_ktime_get_ns();
    memcpy(rec.data, arg, arglen);
    /* submit to per-cpu mmap'd ring buffer in user space */
    u32 key = bpf_get_smp_processor_id();
    bpf_perf_event_output(req, &buf, &key, &rec, sizeof(rec));
}

int read_extension(extfuse_req_t req, fuse_ino_t ino,
        const struct fuse_read_in *in) {
    imap_key_t key = in->fh; /* forward to FUSE daemon if
                on-the-fly data reconstruction is needed */
    if (!extfuse_lookup_imap(&key)) return UPCALL;
    /* trace this I/O request */
    trace_io(req, ino, FUSE_READ, in, sizeof(*in));
    return PASSTHRU; /* forward req to lower FS */
}
```

Figure 5.4: Kernel extension that traces I/O reqs.

5.9.3 Continuous monitoring.

Our ExtFUSE kernel extensions log relevant file system events. Figure 5.4 shows the code. Compared to existing monitoring mechanisms (e.g., inotify) that pose high memory overhead due to recursive watchpoints on each dir, our approach is more efficient since traces are collected inside the kernel as and when file system events are generated. Moreover, memory-mapped I/O can only be traced at the file-system level since no read() or write() system calls are involved.

Table 5.3: Multiple storage management techniques employed by ANODYNE.

Profile	Reclamation	Description
Pristine	`None`	Without ANODYNE
Default	`Deduplication`	Deduplicate files
CleanTemp	`Deletion`	Delete caches and other temporary files
InterpretJavaCode	`Adaptation`	Optimized (AOT) compilation of DEX files
ZippedJavaCode	`Compression`	Extract Libs/DEX (Interpret) from APK

ANODYNE uses Google Play Store services APIs to determine user's location, calendar appointments, and user's physical activity (e.g., walking, running, cycling, etc.)

To evaluate typical size of daily and hourly logs, we utilize data logs collected from the user study. Our results show that we collect on average of compressed 40M of log data daily.

5.10 Evaluation

Fist, we report accuracy of the prediction agent in ANODYNE using the data we collected from 140 users §3.5. We then run micro-benchmarks to evaluate the resource consumption and latency overhead under misprediction for various storage reclamation techniques applied by ANODYNE. Specifically, we measure: a) app launch latency, b) battery, c) network, d) CPU, e) storage, and f) memory consumption during app launch. Finally, we report numbers on storage savings achieved by ANODYNE.

5.10.1 Micro benchmarks

We carry out experiments to measure cold performance of apps under various and compare with that of a pristine system to determine overhead imposed by ANODYNE. Although, in cases where app storage has been reclaimed, it will incur slowdown throughout its usage session depending upon the functionality, we believe app startup performance provides a good estimate of expected slowdown.

Since ANODYNE is a storage service, we also evaluate its performance under no storage reclamation. Table 5.3 lists all configurations we evaluate. All experiments have been

carried out on a Nexus 5 device, running Android Marshmallow (version 6.0.1, API 23). We carry out experiments with the same set of thirty most popular apps The same set of apps were used in our install-time study Figure 3.9.

Methodology. We first install a fresh copy of all the apps. Then each app is started once manually so that it can perform necessary initialization (e.g., account login, fetch and cache data, etc.) since this process may take longer than subsequent app launches. Once all the apps have been initialized, we reboot the device and perform scripted launch of each app 3 times and report average numbers. Previous instances of are force-killed before each run to make sure that cold launch number are captured. App launch time reported here is simply the total time taken to display the app main activity as measured by Android `ActivityManager` framework. The power measurements are taken using Trepen power profiling app from Qualcomm [109] on Nexus 5, which as has a 3.8 V, 2300 mAh LiOn battery offering a total of 8.74 watt hours.

Figure 5.5, Figure 5.6, Figure 5.7 show our results from the experiments. Figure 5.6 shows the CDF of cold app launch times as measured under various settings. As seen from the graph, the launch latency of configuration with runtime interpretation of Java DEX classes is almost similar to the launch latency of system without ANODYNE. The performance can be attributed to heavy EXTFUSE file system optimizations and light-weight continuous profiling that went into designing the system. There is almost negligible performance penalty compared to the pristine (no ANODYNE) system. This is because of our EXTFUSE pass-through file system implementation simply forwards I/O requests targeting at files that are not stubs directly to the native file system, obviating the need for an up-call to the user daemon. Also, we see significant storage savings, but almost no-launch latency for the runtime interpretation config. In contrast, more storage savings but at the cost of high latency for compressed Java DEX files. We can see similar behavior for energy consumption under interpret and compression modes in Figure 5.5.

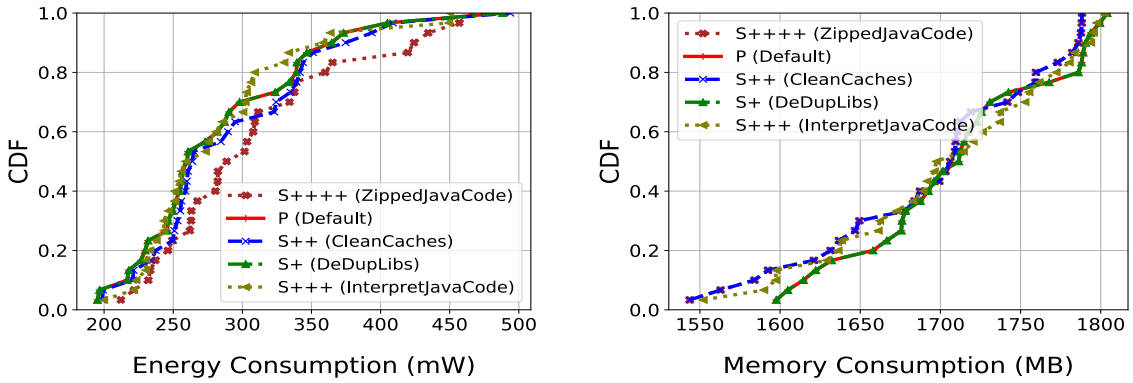


Figure 5.5: Memory and battery overhead from top thirty apps.

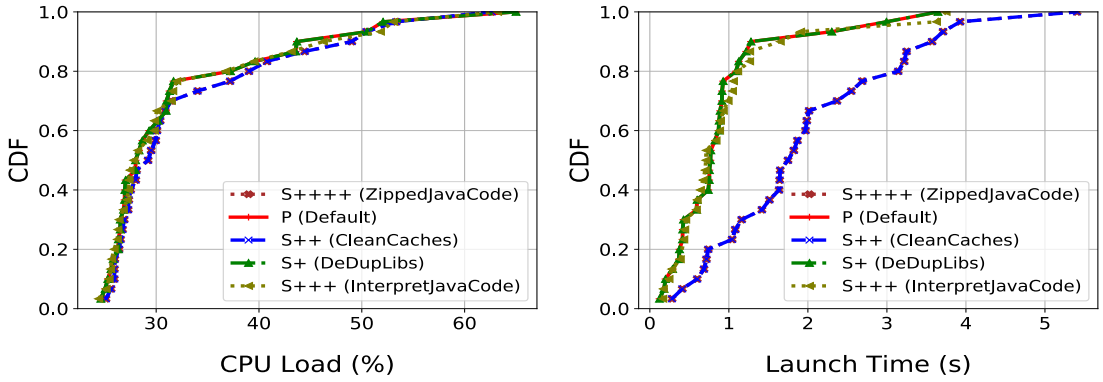


Figure 5.6: CPU and latency overhead from top thirty apps.

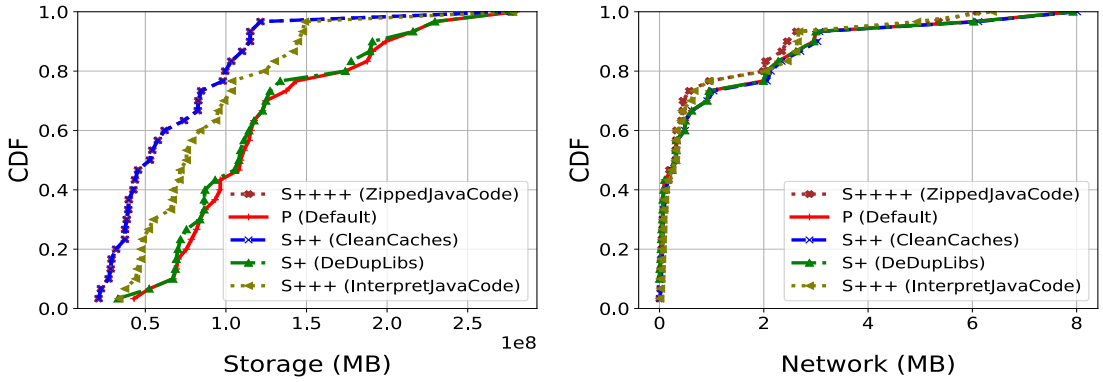


Figure 5.7: Storage and network overhead from top thirty apps.

5.10.2 Storage Savings

As seen from Figure 5.7, ANODYNE can achieve storage savings under various configurations, such deleting temporary files to reclaim cached space and generating adaptive OAT file to reclaim unnecessary storage space occupied by unused app Java classes. We applied multiple reclamation techniques as shown in Table 5.1. Our results show that ANODYNE can achieve at least 60% of storage savings per app. For top 30 apps with an APK average installation size of 47 MB and minimal usage (i.e., one-time launch with negligible app data or cache size), it can save up to 45 MB per app, on average. Nevertheless, this comes at a small cost of increased app latency, network, and cellular resources as shown in Figure 5.5 and Figure 5.6.

5.10.3 Prediction Accuracy

Based on the collected data, we build a storage prediction agent and evaluate its performance. We predict storage needs at two granularity levels, namely per-day prediction and per-hour. We use the data we collected across multiple days from our user study. For each user in our dataset, we select random 25% of days as the test set and use the rest for training. To reduce the influence of test-train partition, we perform random partition three times, and report the average evaluation metrics for our model. We filter out those apps in testing set that do not appear in training set. The batch for all datasets was chosen to be 1 day of app usage.

We experiment with different machine learning models and different feature sets to perform the prediction. The models include Logistic Regression (LOGIT), Linear Regression (LR), Naive Bayes (NB), Support Vector Machine (SVC).

The predictions from our model could be due to false negative (misses) and false positives (over-predictions), which incurs different costs. Therefore, for evaluation metrics, we measure both, Over Prediction Ratio (OPR) and Misprediction Ratio (MPR) to define the overall accuracy since each incur different costs. The OPR/MPR of Naive Bayes (NB) is the base model, and is simply uses the day feature.

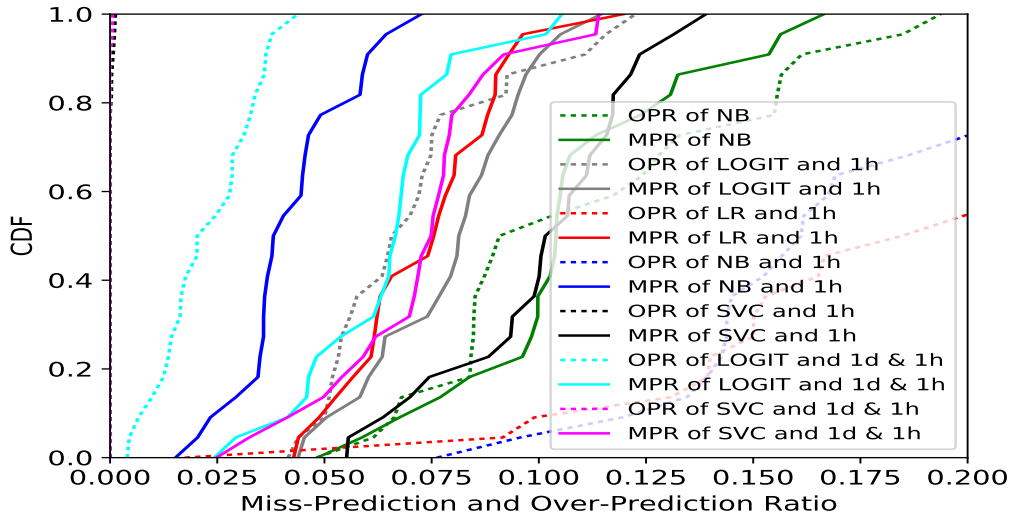


Figure 5.8: **Per-day prediction accuracy.** Miss Prediction and Over Prediction ratios of Per-Day based Prediction for 23 tested users

Per-day Prediction. We consider time, day, location, and app usage history, to predict app usage for the next day. This is a multi-response prediction problem, because the system needs to predict usage for each possible app. The feature sets includes look-back one day and look-back one day + one week for location and app usage. The results is shown in Figure 5.8. Our results suggest that **MPR/OPR of SVC and 1d** is the best model, because it have low miss-prediction ratio and over-prediction ratio.

Per-hour Prediction. Similarly, Figure 5.9 shows the per-hour prediction results. In pre-hour prediction, we look at hour of the day as well as user physical activity (e.g., running, driving, etc.) in addition to the features used in our per-day prediction model. Furthermore, in the per-hour prediction model, we include per-day prediction results assuming that the predictive set of next day apps have already been loaded on the device. Based on these results, we compute the miss-prediction ratio and over-prediction of per-hour prediction, and aggregate the results to a per-day level because we assume that users charge their phones overnight, rather than during the day. According to Figure 5.9, **OPR/MPR of SVC and 1d & 1h** is the best model, because it rarely incurs any over-prediction, and achieves good miss-prediction ratio.

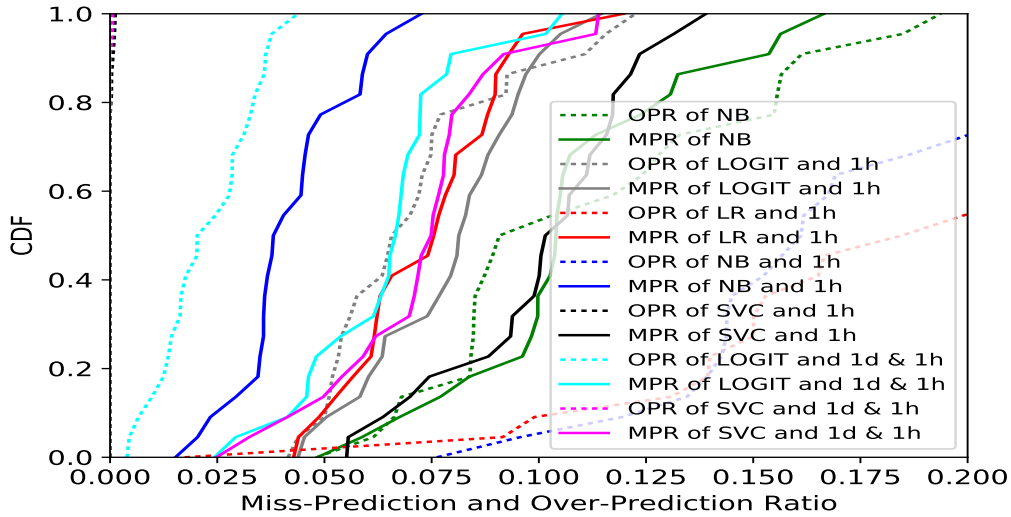


Figure 5.9: **Per-hour prediction accuracy.** Miss Prediction and Over Prediction ratios of Per-Hour based Prediction for 23 tested users

CHAPTER 6

DISCUSSION

Resource consumption. Ideal state to perform storage management tasks is when the device is idle, connected to WiFi, and being charged. However, if ANODYNE is not able to detect such device state, it can also notify users of pending management tasks. Such notifications are currently used to make users aware of pending device backups. Similar context-sensitive management approach can be applied to other resource consumption issues on mobile (battery, memory)

Instant Apps. ANODYNE can benefit from *Instant Apps* [110]; they are accessed using URLs. To enable instant apps, developers modularize their apps. Therefore, when an app module is predicted to be used, ANODYNE can fetch it from the Cloud using its URL, thereby saving additional storage space.

Personal assistant. We extend personal assistant on the device for context-aware automated resource management. This also unlocks a futuristic smartbot interface, whereby the user simply interacts with the assistant on the device using voice commands, but the device itself and its data are completely auto-managed, thereby relieving the user.

Security and Privacy aware management. Since mobile devices are prone to theft and are likely to be lost, it is unsafe to keep sensitive data, such as personal tax documents, bank statements, or work-related confidential data on the device at all times, specially in unencrypted form. ANODYNE can provide additional level of protection. Once apps hosting such sensitive data become inactive (e.g., user leaving home or work), the system can notify them to securely wipe out all personal data from the device.

Misbehaving apps. Since ANODYNE monitors storage accesses when an app is being used, a misbehaving app can potentially generate fake storage traffic to create a bias and adversely affect the prediction. We believe that developers already have an incentive to write

well-behaved apps to keep app performance and user experience under check. Nevertheless, fine-grained accounting of data, similar to Android accounting, can be built into ANODYNE to discourage such behavior.

CHAPTER 7

RELATED WORK

This work is inspired by a number of past works in the area of operating systems, distributed and mobile file systems, mobile computing, and storage. We present a detailed comparison of our work with relevant related work below.

7.1 Network File Systems

Network File System (NFS), Server Message Block (SMB), Common Internet File System (CIFS), and Apple Filing Protocol (AFP) are all network file systems designed for LANs, assuming constant network connectivity and predictable latency. Contrarily, this work targets mobile devices that are personal consumer devices with intermittent network connectivity and limited storage.

7.2 Distributed File Systems

Andrew File System (AFS) [56], Coda [55], and InterMezzo [111] use persistent caching on clients. While AFS caches to improve performance and reduce server traffic, Coda pioneered the idea of local caching (or data hoarding) for disconnected access [112]. The caching policies used, however, are simple and fixed: whole files are cached during `open()` operation. Later versions of AFS overcome this problem by caching partial files. However, the user must manually define custom caching policies through an interactive platform [113].

Seer [114] uses multiple heuristics, such as time-independent semantic distance between files, naming conventions, and directory membership to identify related files and group them into clusters for automated hoarding. Analytic Spy Utility [115] observed file system activity and attempted to build hoard contents based on the structure of the process tree and

the sequence of files accessed. Collected contents are then presented through a GUI to the user to finalize. However, these heuristics assume users work with projects and hoard whole files in a cluster. Mobile devices host disparate data, rendering single or static heuristics not very useful.

This work, on the other hand, aims for predictive and automatic storage management, which not only entails optimal caching, but also efficient compression, deduplication, and data adaptation. ANODYNE proposes multiple storage management techniques to depending on the data type.

7.3 Other Relevant Storage Systems

PersonalRaid [116] and Lookaside Caching [117] explored opportunistic use of mobile devices as a means to cache data to improve performance and availability of distributed file systems. BlueFS [118] is a distributed file system that is designed to minimize storage power consumption on mobile devices. It extends Coda to build a cache hierarchy by storing data on multiple portable devices, which are adaptively accessed to serve data based on energy and performance characteristics.

In this work, we leverage personal Cloud storage for hierarchical management of local storage on mobile devices. Nevertheless, this work can easily be extended to build a similar storage hierarchy by incorporating portable storage (e.g., external sdcard) or peer mobile devices (e.g., phone and tablet) to optimize performance and energy attributes of the system.

RFS [119] is a client-centric network file system that uses Cloud server for mobile client devices. Being targeted for mobile devices, ANODYNE shares some design goals with RFS, namely application-transparency and Cloud augmentation for hierarchical storage management. However, RFS does not manage storage. As such, respective end goals are very different.

7.4 Data Replication Systems

Eyo [120], Segank [121], EnsemBlue [122], PRACTI [123], Bayou [124], Cimbiosys [125], Perspective [126], and Footloose [127] focus on data replication and consistency across personal devices. Eyo aims for device-transparent storage through use of API. Similar to the proposed work, Eyo also targets storage-limited devices and supports offline access through replication. However, it fully replicates metadata and policies across all devices, but uses application-specific placement rules for data replication. Our goal, on the other hand, is to support application-transparency. EnsemBlue optimizes BlueFS for peer-to-peer updates. It uses simple content-aware and affinity-based rules to replicate data. Similarly, Perspective supports view query that defines file-device mapping based on metadata tags. PRACTI and Bayou provide a weakly-connected replicated storage. None of above-mentioned systems provide intelligent storage management.

Ori File System (OriFS) [75] is a distributed file system that leverages data versioning to maintain entire file system history. It also supports offline data access. However, it does not consider storage space constraints on mobile devices and uses complete replication.

Distributed version control systems such as Git [128] provide offline access to content. However, they are mainly for source versioning control and do not provide file system-like interface to users. Git Annex Assistant [129] provides web-based interface and a mobile application for keeping user data in sync on all the devices. However, it does not support automatic storage management or allows offline data organization.

7.5 Data Adaptation

Odyssey [130] and quFiles [131] explore data adaptation to deal with energy challenges [132], limited network bandwidth, and data incompatibility issues in distributed file systems. While Odyssey requires application support to adapt functionality with resource consumption, quFiles is application-transparent and relies on context-aware policies to retrieve a particular

data representation. This work too leverages data adaptation, but for efficient use of limited storage on mobile devices.

Pillai et al. [133] looked at applying multi-fidelity adaptations for storage to archive images and videos. Spectra [134] and its extension Chroma [135] use multi-fidelity algorithms [136] to improve energy and latency respectively for remote execution of applications. While Spectra monitors applications for estimated resource consumption to automatically select the appropriate fidelity, Chroma takes a utility-based approach to let applications define latency for more meaningful adaptation. GRACE [137] project proposes cross-layer adaptation to improve energy efficiency. We too leverage content adaptation for optimal use of mobile storage. For example, we selectively pre-compile DEX bytecode to machine-executable OAT files to carefully balance runtime performance, energy consumption, and storage. We choose appropriate adaptation based on content type and its latency and resource requirements.

7.6 Storage Management

Harmonium [138] introduced motif abstraction for storage management that allows apps to register handlers for reconstructing data on-demand by decompressing or fetching over the network. This work too leverages compression and Cloud-backed hierarchical management to reclaim storage space. However, it proposes predictive and application-transparent storage management.

Elastic quota [22] manages storage by hard-limiting only persistent data and and allowing temporary data to grow or be reclaimed depending upon the available storage space. However, such a design requires the user to identify when and what data should persist or thrown away. We propose context-sensitive storage quota that infers temporary data based on active user context for automatic management.

RedDroid [139] performs static analysis of Android apps to remove code bloat. Like RedDroid, we also carry out static analysis of mobile apps to report source of bloating from

storage consumption point of view.

7.7 Context-aware Computing and Prefetching

There has been a substantial research on the use of context awareness and prefetching to various domains, including file systems [140], storage [141], operating systems, and mobile computing to improve performance.

Prefetching based on related file access pattern is a common technique to improve performance of distributed file systems [142, 143, 144, 145]. This work also proposes continuous storage and device monitoring to identify storage needs of the user for timely prefetching. Our user study shows that only a small fraction of apps are actively used. We propose to infer high-level storage needs based usage access patterns.

On mobile devices, several studies have looked at using contextual information for application prediction for better manageability of apps on the home screen [146, 101, 103], reduce launch latency [99], target ads [147], and data prefetching [100, 148]. Nevertheless, informed prefetching [140, 148] proposes to modify apps to convey prefetching hints to the system. These studies have shown that app usage on mobile devices depend heavily on app usage history and contextual attributes, such as location of the user, time of the day, sensor (e.g., accelerometer), and WiFi/Cellular status. In this work, we explore the use of contextual information for automated and application-transparent storage management on mobile devices.

7.8 Cloud Storage Services

Existing commercial Cloud storage solutions, such as Dropbox [13], Box [14], iCloud [16], and Google drive [15] primarily provide data sharing, synchronization, and backup services for ubiquitous and on-demand access to user data. However, they do not support automatic storage management, which is the focus of this work. Users perform manual cleanup to reclaim space after backing up their documents, pictures, and videos. While services such as

Google Photos [149] can automatically backup photos and safely delete them after backup, it currently only supports photos and videos. The only way users reclaim space occupied by apps is by deleting them. iOS v11 provides an optional feature to automatically offload infrequently used apps to iCloud [17], but it only removes the core app while retaining all its settings and data (e.g., login credentials) for future stateful accesses. While Android users can forcefully clear app caches and data as needed, iOS offers no way for users to delete app data without uninstalling the app [18].

MultiCloud [150] approaches unify multiple Cloud storage solutions to provide a centralized interface for easy management and cumulative storage. ANODYNE can be easily extended to integrated multiple Cloud storage solutions for centralized hierarchical storage management.

S3FS [151] is a network file system built on top of Amazon's S3 Cloud storage. It mounts Cloud storage as a local partition under the host file system. However, it is not resource efficient and lacks storage management feature.

7.9 User File System Frameworks

There exists a number of frameworks to develop user file systems. Here we compare them with our EXTFUSE framework.

A number of user file systems have been implemented using NFS loopback servers [152]. UserFS [69] exports generic VFS-like file system requests to the user space through a file descriptor. Arla [66] is an AFS client system that lets apps implement a file system by sending messages through a device file interface `/dev/xfs0`. Coda file system [55] exported a similar interface through `/dev/cfs0`. NetBSD provides Pass-to-Userspace Framework FileSystem (PUFFS). Mazières et al. proposed a C++ toolkit that exposes a NFS-like interface for allowing file systems to be implemented in user space [68]. UFO [67] is a global file system implemented in user space by introducing a specialized layer between the apps and the OS that intercepts file system calls.

FUSE. File System Translator (FiST) [153] is a tool for simplifying the development of stackable file system. It provides *boilerplate* template code and allows developers to only implement the core functionality of the file system. FiST does not offer safety and reliability as offered by user space file system implementation. Additionally, it requires learning a slightly simplified file system language that describes the operation of the stackable file system. Furthermore, it only applies to stackable file systems.

Narayan et al. [154] combined in-kernel stackable FiST driver with FUSE to offload data from I/O requests to user space to apply complex functionality logic and pass processed results to the lower file system. Their approach is only applicable to stackable file systems. They further rely on static per-file policies based on extended attributes labels to enable or disable certain functionality. In contrast, EXTFUSE downloads and safely executes thin extensions from user file systems in the kernel that encapsulate their rich and specialized logic to serve requests in the kernel and skip unnecessary user-kernel switching.

7.10 Extensible Systems

Past works have explored the idea of letting apps extend system services at runtime to meet their performance and functionality needs. SPIN [155] and VINO [156] allow apps to safely insert kernel extensions. SPIN uses a type-safe language runtime, whereas VINO uses software fault isolation to provide safety. ExoKernel [157] is another OS design that lets apps define their functionality. Systems such as ASHs [158, 159] and Plexus [160] introduced the concept of network stack extension handlers inserted into the kernel. SLIC [161] extends services in monolithic OS using interposition to enable incremental functionality and composition. SLIC assumes that extensions are trusted. EXTFUSE is a framework that allows user file systems to add "thin" extensions in the kernel that serve as specialized interposition layers to support both in-kernel and user space processing to co-exist in monolithic OSes.

eBPF. This works is not the first one to use eBPF for safe extensibility. eXpress DataPath

(XDP) [162] allows apps to insert eBPF hooks in the kernel for faster packet processing and filtering. Amit et al. proposed Hyperupcalls [163] as eBPF helper functions for guest VMs that are executed by the hypervisor. More recently, SandFS [164] uses eBPF to provide an extensible file system sandboxing framework. Like ExtFUSE, it also allows unprivileged apps to insert custom security checks into the kernel.

7.11 Study of Mobile apps stores

The following works compare to our study of mobile apps.

App store study. PlayDrone [29] was the first work to extensively study the Google Play Store. [165] carried out a large-scale study of file systems. [166] studied app download patterns, popularity trends, and development strategies in mobile app ecosystem. [167] studied usage patterns of smart mobile devices.

Various studies on mobile apps have also been carried out to identify known (n-day) security vulnerabilities. For example, OSSPolice [30] identified license violations and security issues with Open source software used in mobile apps. Similarly, LibScout [33] studied security issues with Java third-party libraries. Both the tools detect libraries and correlate them with existing vulnerability data to identify vulnerable ones. In contrast, we carry out the first large-scale study on storage consumption of mobile apps.

CHAPTER 8
CONCLUSION

In this work, we presented the first ever detailed measurement study of storage consumption behavior of millions of mobile apps. Our longitudinal analysis shows that modern mobile apps have evolved as large monolithic packages. Developers create universal apps, packing more features than ever. Today, an app can consume up to 4 GB of storage space. While it clearly signifies the evolution of mobile app ecosystem in terms of features, it also implies heavy future storage demands. We believe this trend will continue as the mobile technology evolves (e.g., 5G) and richer, more immersive apps supporting Augmented Reality, Artificial Intelligence, and 4K graphics are introduced.

While users today can pay a premium price for additional storage, our user study shows that modern apps are bloated with features and heavily optimized for performance by default at the cost of more storage. Users typically use only a fraction of apps and features actively, and app usage is highly correlated to the user context. As such, storing all app features in a performance-optimized manner at all times is neither desirable nor required. Furthermore, developers freely abuse persistent storage to cache data for streamlined user experience, host ads and user analytics for monetization.

Based on our study findings, in this work we further introduce a context-aware automated storage management architecture, called ANODYNE, for smart mobile devices. The ANODYNE system tracks everyday storage needs and automatically builds working set profiles for multiple contexts, such as location, day/time, and calendar. The system then reclaims contextually unwanted apps/data using multiple storage management techniques, such as deduplication, selective compression, deletion, content adaptation, and Cloud-backed hierarchical management. Reclaimed data is reconstructed predictively based on the active user context or on demand under misprediction. Data reconstruction is performed by either

computation on the device (e.g., decompression) or fetching from the Cloud. The evaluation of our prototype system on Android with top 30 apps show that ANODYNE saves a minimum of 60% of storage space occupied by app APKs. At the same time, it imposes little to no overhead when no data reconstruction is required.

REFERENCES

[1] AppBrain, *Number of android applications*, 2020.

[2] G. Inc., *Android market client update — android developers blog*, Dec. 2010.

[3] ——, *Android apps break the 50mb barrier — android developers blog*, Mar. 2012.

[4] A. Inc., *Now accepting larger binaries - news and updates - apple developer*, Feb. 2015.

[5] TmoNews, *Google head of android user experience explains the lack of sd cards for nexus devices.* Oct. 2012.

[6] SanDisk, *62 percent of indians run out of smartphone space every 3 months: Sandisk*, Mar. 2018.

[7] H. Post, *Apple sued over storage-devouring ios 8.* Dec. 2014.

[8] P. McDaniel, "Bloatware comes to the smartphone," *IEEE Security & Privacy*, no. 4, pp. 85–87, 2012.

[9] ZDNet, *How much free storage space does your smartphone really have?* Jan. 2015.

[10] G. Inc., *Faster storage statistics*, Sep. 2019.

[11] U. Today, *Ios8 users massively deleting to make room.* Sep. 2014.

[12] A. Singla, *The mobile app industry's worst-kept secret*, Jan. 2017.

[13] Dropbox Inc., *Dropbox - your stuff, anywhere.* Feb. 2015.

[14] Box Inc., *Box — free cloud storage, secure content & online file sharing service*, Sep. 2019.

[15] G. Inc., *Google drive - cloud storage & file backup for photos, docs & more*, Sep. 2019.

[16] Apple Inc., *Icloud*, Feb. 2019.

[17] A. Inc., *If you need more space for an update*, Sep. 2019.

[18] A. T. Inc., *How to clear the cache on your iphone or ipad*, Jul. 2017.

[19] Wikipedia, *Iphone xs*, Apr. 2019.

[20] G. Inc., *About android app bundles*, Sep. 2019.

[21] ——, *Multiple apk support*, Sep. 2019.

[22] E. Zadok, J. Osborn, A. Shater, C. P. Wright, K. Muniswamy-Reddy, and J. Nieh, "Reducing storage management costs via informed user-based policies," in *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST)*, College Park, MD, USA, Apr. 2004.

[23] *Android abis*, Feb. 2019.

[24] Google Inc., *Apk expansion files — android developers*, Sep. 2019.

[25] G. Inc., *Partitions and images — android open source project*, Sep. 2019.

[26] Ext4, *Ext4 wiki*, Mar 2015.

[27] A. Developers, *Android 3.0 apis*, Mar. 2015.

[28] M. Szeredi and N.Rauth, *Fuse - filesystems in userspace*, 2018.

[29] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *Proceedings of the 2014 ACM SIGMETRICS Conference*, Austin, TX, Jun. 2014.

[30] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, Texas: ACM, Oct. 2017.

[31] Google Inc., *Support for 100mb apks on google play — android developers blog*, Sep. 2015.

[32] L. Richardson, "Beautiful soup."

[33] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[34] *Support different languages and cultures*, Feb. 2019.

[35] The Apache Software Foundation, *Apache maven project*, 2016.

[36] Bintray.com, *Jcenter is the place to find and share popular apache maven packages*, 2016.

[37] G. Inc., *Reduce apk size*, Sep. 2019.

[38] NetworkWorld, *Intel confronts a big mobile challenge: native compatibility*, 2014.

[39] G. Inc., *Data and file storage overview*, Dec. 2019.

[40] A. Nandugudi, A. Maiti, T. Ki, F. Bulut, M. Demirbas, T. Kosar, C. Qiao, S. Y. Ko, and G. Challen, "Phonelab: A large programmable smartphone testbed," in *Proceedings of First International Workshop on Sensing and Big Data Mining*, ser. SENSEMINE'13, Roma, Italy, 2013, 4:1–4:6.

[41] J. Shi, E. Santos, and G. Challen, *Lessons from four years of phonelab experimentation*, 2019. arXiv: `1902.01929 [cs.NI]`.

[42] G. Inc., *Android logcat system*, Jul. 2020.

[43] R. Love, "Kernel korner: Intro to inotify," *Linux Journal*, vol. 2005, p. 8, 2005.

[44] J. Corbet, *Superblock watch for fsnotify*, Apr. 2017.

[45] R. Flament, *Loggedfs - filesystem monitoring with fuse*, Mar. 2018.

[46] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To fuse or not to fuse: performance of user-space file systems," in *15th USENIX Conference on File and Storage Technologies (FAST) (FAST 17)*, Santa Clara, CA: USENIX Association, Feb. 2017.

[47] Wikipedia, *Iphone 7*, Feb. 2015.

[48] C. Research, *Average storage capacity in smartphones to cross 80gb by end-2019*, Mar. 2019.

[49] IDC, *Smartphone market share*, Dec. 2019.

[50] Statcounter, *Mobile operating system market share india*, Dec. 2019.

[51] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for unix," *ACM Trans. Comput. Syst.*, vol. 2, pp. 181–197, Aug. 1984.

[52] R. Card, "Design and implementation of the second extended filesystem," in *Proceedings of the First Dutch International Symposium on Linux*, 1995.

[53] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system," in *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, San Diego, CA, Jan. 1996, pp. 1–1.

[54] B. Nowicki, *Nfs: network file system version protocol specification*, Mar. 1989.

[55] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, Apr. 1990.

[56] J. H. Howard, "An overview of the andrew file system," *In Winter 1988 USENIX Conference Proceedings*, pp. 23–26, 1988.

[57] C. P. Wright, M. Martino, and E. Zadok, "NCryptfs: a secure and convenient cryptographic file system," in *Proceedings of the 2003 USENIX Annual Technical Conference (ATC)*, San Antonio, TX, Jun. 2003.

[58] A. Aranya, C. P. Wright, and E. Zadok, "Tracefs: a file system to trace them all," in *3rd USENIX Conference on File and Storage Technologies (FAST) (FAST 04)*, San Francisco, CA: USENIX Association, Mar. 2004.

[59] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Iron file systems," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, Oct. 2005.

[60] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for unix development," pp. 93–112, 1986.

[61] J. Liedtke, "Improving ipc by kernel design," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, NC, Dec. 1993, pp. 175–188.

[62] *Gnu hurd*, Apr. 2018.

[63] S. Sundararaman, L. Visampalli, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Refuse to crash with re-fuse," in *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, Salzburg, Austria, Apr. 2011.

[64] J. H. Hartman and J. K. Ousterhout, "Performance measurements of a multiprocessor sprite kernel.," in *Proceedings of the Summer 1990 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, 1990, pp. 279–288.

[65] D. C. Steere, J. J. Kistler, and M. Satyanarayanan, "Efficient user-level file cache management on the sun vnode interface," in *Proceedings of the Summer 1990 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, 1990.

[66] A. Westerlund and J. Danielsson, "Arla: A free afs client," in *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*, New Orleans, Louisiana, Jun. 1998, pp. 32–32.

[67] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman, "Extending the operating system at the user level: the ufo global file system," in *Proceedings of the 1997 USENIX Annual Technical Conference (ATC)*, Anaheim, California, Jan. 1997, pp. 6–6.

[68] D. Mazières, "A toolkit for user-level file systems," in *Proceedings of the 2001 USENIX Annual Technical Conference (ATC)*, Jun. 2001, pp. 261–274.

[69] J. Fitzhardinge, *Userfs*, Mar. 2018.

[70] A. Kantee, "Puffs-pass-to-userspace framework file system," in *Proceedings of the Asian BSD Conference (AsiaBSDCon)*, Tokyo, Japan, Mar. 2007.

[71] Ungureanu, Cristian and Atkin, Benjamin and Aranya, Akshat and Gokhale, Salil and Rago, Stephen and Całkowski, Grzegorz and Dubnicki, Cezary and Bohra, Aniruddha, "Hydrafs: a high-throughput file system for the hydrastor content-addressable storage system," in *8th USENIX Conference on File and Storage Technologies (FAST) (FAST 10)*, San Jose, CA: USENIX Association, Feb. 2010, pp. 17–17.

[72] B. Cornell, P. A. Dinda, and F. E. Bustamante, "Wayback: A user-level versioning file system for linux," in *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jun. 2004, pp. 27–27.

[73] R. Pontes, D. Burihabwa, F. Maia, J. a. Paulo, V. Schiavoni, P. Felber, H. Mercier, and R. Oliveira, "Safefs: A modular architecture for secure user-space file systems: One fuse to rule them all," in *Proceedings of the 10th ACM International on Systems and Storage Conference*, Haifa, Israel, May 2017, 9:1–9:12.

[74] K. Ren and G. Gibson, "Tablefs: Enhancing metadata efficiency in the local file system," in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, San Jose, CA, Jun. 2013, pp. 145–156.

[75] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières, "Replication, history, and grafting in the ori file system," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[76] H. Sigurbjarnarson, P. O. Ragnarsson, J. Yang, Y. Vigfusson, and M. Balakrishnan, "Enabling space elasticity in storage systems," in *Proceedings of the 9th ACM International on Systems and Storage Conference*, Haifa, Israel, Jun. 2016, 6:1–6:11.

[77] *Storage — android open source project*, Sep. 2018.

[78] O. Z. Community, *Zfs on linux*, Apr. 2018.

[79] *Gluster*, Apr. 2018.

[80] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006, pp. 307–320.

[81] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To fuse or not to fuse: performance of user-space file systems," in *15th USENIX Conference on File and Storage Technologies (FAST) (FAST 17)*, Santa Clara, CA: USENIX Association, Feb. 2017.

[82] Ceph, *Ceph kernel client*, Apr. 2018.

[83] Gluster, *Libgfapi*, Apr. 2018.

[84] *Ebpf: extended berkley packet filter*, 2017.

[85] E. Zadok, I. Bădulescu, and A. Shender, "Extending file systems using stackable templates"," in *Proceedings of the 1999 USENIX Annual Technical Conference (ATC)*, Jun. 1999, pp. 57–70.

[86] N. Rauth, *A network filesystem client to connect to ssh servers*, Apr. 2018.

[87] L. . GitHub, *Without 'default_permissions', cached permissions are only checked on first access*, 2018.

[88] *A featureful union filesystem*, Mar. 2018.

[89] M. Pärtel, *Bindfs*, 2018.

[90] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *;login: The USENIX Magazine*, vol. 41, no. 1, pp. 6–12, Mar. 2016.

[91] E. Database, *Android - sdcardfs changes current-¿fs without proper locking*, 2019.

[92] 96boards, *Hikey (lemaker) development boards*, May 2019.

[93] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *Proceedings of the 8th ACM International Conference on Mobile Computing Systems (MobiSys)*, San Francisco: Association for Computing Machinery, Inc., Jun. 2010, pp. 179–194.

[94] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage," in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, ser. MobileHCI '11, Stockholm, Sweden: ACM, 2011, pp. 47–56.

[95] T. Crunch, *Users have low tolerance for buggy apps only 16% will try a failing app more than twice*, Mar. 2013.

[96] G. Inc., *Keeping your app responsive — android developers*, Dec. 2018.

[97] ——, *Data and file storage overview — android developers*, Mar. 2019.

[98] ——, *Back up user data with auto backup*, Mar. 2019.

[99] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, ACM, 2012, pp. 113–126.

[100] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin, "Practical prediction and prefetch for faster access to applications on mobile phones," in *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, Zurich, Switzerland: ACM, Sep. 2013, pp. 275–284.

[101] K. Huang, C. Zhang, X. Ma, and G. Chen, "Predicting mobile application usage using contextual information," in *Proceedings of the 2012 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, Pittsburgh, Pennsylvania: ACM, Sep. 2012, pp. 1059–1065.

[102] C. Shin, J.-H. Hong, and A. K. Dey, "Understanding and prediction of mobile application usage for smart phones," in *Proceedings of the 2012 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, Pittsburgh, Pennsylvania: ACM, Sep. 2012, pp. 173–182.

[103] R. Baeza-Yates, D. Jiang, F. Silvestri, and B. Harrison, "Predicting the next app that you are going to use," in *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, Shanghai, China, Jan. 2015, pp. 285–294.

[104] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD)*, Portland, Oregon, Aug. 1996, pp. 226–231.

[105] O. Foundation, *Nominatim - openstreetmap wiki*, Mar. 2019.

[106] J. H. Kang, W. Welbourne, B. Stewart, and G. Borriello, "Extracting places from traces of locations," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, no. 3, pp. 58–68, 2005.

[107] A. Bijlani and U. Ramachandran, "Extension framework for file systems in user space," in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, Jul. 2019, pp. 121–134.

[108] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: A retrospective," in *Proceedings of the 2011 Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, 2011.

[109] G. Inc., *Trepn power profiler - qualcomm developer network*, Feb. 2018.

[110] ——, *Android instant apps — android developers*, Feb. 2018.

[111] P. J. Braam, M. Callahan, and Schwan, "The intermezzo file system," *The Perl Conference 3*, Aug. 1999.

[112] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 3–25, 1992.

[113] M. R. Ebling, B. E. John, and M. Satyanarayanan, "The importance of translucence in mobile computing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 9, no. 1, pp. 42–67, Mar. 2002.

[114] G. H. Kuenning and G. J. Popek, "Automated hoarding for mobile computers," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, Oct. 1997.

[115] C. Tait, H. Lei, S. Acharya, and H. Chang, "Intelligent file hoarding for mobile computers," in *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking (MobiComm)*, Berkeley, California, USA, Nov. 1995.

[116] S. Sobti, N. Garg, A. Krishnamurthy, C. Zhang, X. Yu, and R. Y. Wang, "Personal-raid: Mobile storage for distributed and disconnected computers," in *1st USENIX Conference on File and Storage Technologies (FAST) (FAST 02)*, San Francisco, CA: USENIX Association, Mar. 2002.

[117] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan, "Integrating portable and distributed storage," in *3rd USENIX Conference on File and Storage Technologies (FAST) (FAST 04)*, San Francisco, CA: USENIX Association, Mar. 2004.

[118] E. B. Nightingale and J. Flinn, "Energy-efficiency and storage flexibility in the blue file system," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[119] Y. Dong, H. Zhu, J. Peng, F. Wang, M. P. Mesnier, D. Wang, and S. C. Chan, "Rfs: A network file system for mobile devices and the cloud," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 101–111, Feb. 2011.

[120] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. T. Morris, and M. F. Kaashoek, "Eyo: Device-transparent personal storage.," in *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, Portland, OR, Jun. 2011, p. 35.

[121] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Y. Wang, "Segank: A distributed mobile storage system," in *3rd USENIX Conference on File and Storage Technologies (FAST) (FAST 04)*, San Francisco, CA: USENIX Association, Mar. 2004.

[122] D. Peek and J. Flinn, "Ensemblue: Integrating distributed storage and consumer electronics," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[123] N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "Practi replication.," in *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006, pp. 5–5.

[124] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995, 172182.

[125] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat, "Cimbiosys: A platform for content-based partial replication," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Apr. 2009, 261276.

[126] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger, "Perspective: Semantic data management for the home," in *7th USENIX Conference on File and Storage Technologies (FAST) (FAST 09)*, San Jose, CA: USENIX Association, Feb. 2009.

[127] J. M. Paluska, D. Saff, T. Yeh, and K. Chen, "Footloose: A case for physical eventual consistency and selective conflict resolution," in *Proceedings of the first ACM International Conference on Mobile Computing Systems (MobiSys)*, San Francisco, CA: Association for Computing Machinery, Inc., May 2003.

[128] S. F. C. Inc., *Git*, Sep. 2015.

[129] J. Hess, *Git-annex branchable.* Sep. 2015.

[130] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile application-aware adaptation for mobility," pp. 245–254, Oct. 1997.

[131] K. Veeraraghavan, J. Flinn, E. B. Nightingale, and B. Noble, "Qufiles: The right file at the right time," in *8th USENIX Conference on File and Storage Technologies (FAST) (FAST 10)*, San Jose, CA: USENIX Association, Feb. 2010.

[132] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawa Island, SC, Dec. 1999.

[133] P. Pillai, Y. Ke, and J. Campbell, "Multi-fidelity storage," in *Proceedings of the ACM 2Nd International Workshop on Video Surveillance &Amp; Sensor Networks*, New York, NY, USA: ACM, 2004, pp. 72–79.

[134] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, Jun. 2002, pp. 217–226.

[135] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi, "Tactics-based remote execution for mobile computing," in *Proceedings of the first ACM International Conference on Mobile Computing Systems (MobiSys)*, San Francisco, CA: Association for Computing Machinery, Inc., May 2003.

[136] M. Satyanarayanan and D. Narayanan, "Multi-fidelity algorithms for interactive mobile applications," *Wirel. Netw.*, vol. 7, no. 6, pp. 601–607, Nov. 2001.

[137] W. Yuan, K. Nahrstedt, S. V. Adve, D. L. Jones, and R. H. Kravets, "Grace-1: Cross-layer adaptation for multimedia quality and battery energy," *IEEE Transactions on Mobile Computing*, vol. 5, no. 7, pp. 799–815, 2006.

[138] H. Sigurbjarnarson, P. O. Ragnarsson, Y. Vigfusson, and M. Balakrishnan, "Harmonium: Elastic cloud storage via file motifs," in *Proceedings of the 6th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Philadelphia, PA: USENIX Association, Jun. 2014.

[139] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu, "Reddroid: Android application redundancy customization based on static analysis.," in *International Symposium on Software Reliability Engineering*, 2018, pp. 189–199.

[140] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[141] C. K. Hess and R. H. Campbell, "A context-aware data management system for ubiquitous computing application," in *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, Providence, Rhode Island, Jun. 2003.

[142] G. H. Kuenning and G. J. Popek, "Automated hoarding for mobile computers," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, Oct. 1997.

[143] T. Kroeger and D. D. E. Long, "The case for efficient file access pattern modeling," in *Proceedings of the 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Mar. 1999, 1419.

[144] A. Amer, D. D. E. Long, and R. C. Burns, "Group-based management of distributed file caches," in *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, Jun. 2002.

[145] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan, "Data staging on untrusted surrogates," in *2nd USENIX Conference on File and Storage Technologies (FAST) (FAST 03)*, San Francisco, CA: USENIX Association, Mar. 2003, pp. 2–2.

[146] X. Zou, W. Zhang, S. Li, and G. Pan, "Prophet: What app you wish to use next," in *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, Zurich, Switzerland: ACM, Sep. 2013, pp. 167–170.

[147] J. P. Rula, B. Jun, and F. Bustamante, "Mobile ad(d): Estimating mobile app session times for better ads," in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, Santa Fe, New Mexico, USA, Feb. 2015, pp. 123–128.

[148] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson, "Informed mobile prefetching," in *Proceedings of the 10th ACM International Conference on Mobile Computing Systems (MobiSys)*, Lake District, United Kingdom: Association for Computing Machinery, Inc., Jun. 2012.

[149] Google Inc., *Google photos - all your photos organized and easy to find*, Sep. 2015.

[150] M. Inc., *Manage, move, copy, and migrate files between cloud*, Sep. 2015.

[151] R. Rizun, *Fuse-based file system backed by amazon s3*, Apr. 2018.

[152] D. K. Gifford, P. Jouvelot, M. A. Sheldon, *et al.*, "Semantic file systems," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, Oct. 1991, pp. 16–25.

[153] E. Zadok and J. Nieh, "FiST: a language for stackable file systems," in *Proceedings of the 2000 USENIX Annual Technical Conference (ATC)*, Jun. 2000.

[154] S. Narayan, R. K. Mehta, and J. A. Chandy, "User space storage system stack modules with file level control," in *Proceedings of the Linux Symposium*, Ottawa, Canada, Jul. 2010, pp. 189–196.

[155] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, safety and performance in the spin operating system," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[156] M. Seltzer, Y. Endo, C. Small, and K. A. Smith, "An introduction to the architecture of the vino kernel," Technical Report 34-94, Harvard Computer Center for Research in Computing Technology, Tech. Rep., Oct. 1994.

[157] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[158] D. A. Wallach, D. R. Engler, and M. F. Kaashoek, "Ashs: application-specific handlers for high-performance messaging," in *Proceedings of the 7th ACM SIGCOMM*, Palo Alto, CA, Aug. 1996.

[159] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney, "Fast and flexible application-level networking on exokernel systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 1, pp. 49–83, 2002.

[160] M. E. Fiuczynski and B. N. Bershad, "An extensible protocol architecture for application-specific networking," in *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, San Diego, CA, Jan. 1996.

[161] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson, "Slic: An extensibility system for commodity operating systems," in *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*, New Orleans, Louisiana, Jun. 1998.

[162] IOVisor, *Xdp - io visor project*, May 2019.

[163] N. Amit and M. Wei, "The design and implementation of hyperupcalls," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jul. 2018, pp. 97–112.

[164] A. Bijlani and U. Ramachandran, "A lightweight and fine-grained file system sandboxing framework," in *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys)*, Jeju Island, South Korea, Aug. 2018.

[165] J. R. Douceur and W. J. Bolosky, "A large-scale study of file-system contents," *SIGMETRICS Perform. Eval. Rev.*, pp. 59–70,

[166] T. Petsas, A. Papadogiannakis, M. Polychronakis, E. P. Markatos, and T. Karagiannis, "Rise of the planet of the apps: A systematic study of the mobile app ecosystem," in *Proceedings of the 2013 conference on Internet measurement conference*, 2013, pp. 277–290.

[167] H. Li, X. Lu, X. Liu, T. Xie, K. Bian, F. X. Lin, Q. Mei, and F. Feng, "Characterizing smartphone usage patterns from millions of android users," in *Proceedings of the 2015 Internet Measurement Conference*, 2015, pp. 459–472.